Chapter 4

THE GREEDY METHOD

4.1 THE GENERAL METHOD

The greedy method is perhaps the most straightforward design technique we consider in this text, and what's more it can be applied to a wide variety of problems. Most, though not all, of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a *feasible* solution. We need to find a feasible solution that either maximizes or minimizes a given *objective function.* A feasible solution that does this is called an *optimal solution.* There is usually an obvious way to determine a feasible solution but not necessarily an optimal solution.

The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added. The selection procedure itself is based on some optimization measure. This measure may be the objective function. In fact, several different optimization measures may be plausible for a given problem. Most of these, however, will result in algorithms that generate suboptimal solutions. This version of the greedy technique is called the *subset paradigm.*

We can describe the subset paradigm abstractly, but more precisely than above, by considering the control abstraction in Algorithm 4.1.

The function Select selects an input from $a[\,]$ and removes it. The selected input's value is assigned to x. Feasible is a Boolean-valued function that determines whether x can be included into the solution vector. The function Union combines x with the solution and updates the objective function. The

```
1   Algorithm Greedy(a, n)
2   // a[1 : n] contains the n inputs.
3   {
4       solution := ∅; // Initialize the solution.
5       for i := 1 to n do
6       {
7           x := Select(a);
8           if Feasible(solution, x) then
9               solution := Union(solution, x);
10      }
11      return solution;
12  }
```

Algorithm 4.1 Greedy method control abstraction for the subset paradigm

function Greedy describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions Select, Feasible, and Union are properly implemented.

For problems that do not call for the selection of an optimal subset, in the greedy method we make decisions by considering the inputs in some order. Each decision is made using an optimization criterion that can be computed using decisions already made. Call this version of the greedy method the *ordering paradigm.* Sections 4.2, 4.3, 4.4, and 4.5 consider problems that fit the subset paradigm, and Sections 4.6, 4.7, and 4.8 consider problems that fit the ordering paradigm.

EXERCISE

1. Write a control abstraction for the ordering paradigm.

4.2 KNAPSACK PROBLEM

Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \tag{4.2}$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximized.

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

Lemma 4.1 In case the sum of all the weights is $\le m$, then $x_i = 1$, $1 \le i \le n$ is an optimal solution. □

So let us assume the sum of weights exceeds m. Now all the x_i's cannot be 1. Another observation to make is:

Lemma 4.2 All optimal solutions will fill the knapsack exactly. □

Lemma 4.2 is true because we can always increase the contribution of some object i by a fractional amount until the total weight is exactly m.

Note that the knapsack problem calls for selecting a subset of the objects and hence fits the subset paradigm. In addition to selecting a subset, the knapsack problem also involves the selection of an x_i for each object. Several simple greedy strategies to obtain feasible solutions whose sums are identically m suggest themselves. First, we can try to fill the knapsack by including next the object with largest profit. If an object under consideration doesn't fit, then a fraction of it is included to fill the knapsack. Thus each time an object is included (except possibly when the last object is included)

into the knapsack, we obtain the largest possible increase in profit value. Note that if only a fraction of the last object is included, then it may be possible to get a bigger increase by using a different object. For example, if we have two units of space left and two objects with ($p_i = 4$, $w_i = 4$) and ($p_j = 3, w_j = 2$) remaining, then using j is better than using half of i. Let us use this selection strategy on the data of Example 4.1.

Object one has the largest profit value ($p_1 = 25$). So it is placed into the knapsack first. Then $x_1 = 1$ and a profit of 25 is earned. Only 2 units of knapsack capacity are left. Object two has the next largest profit ($p_2 = 24$). However, $w_2 = 15$ and it doesn't fit into the knapsack. Using $x_2 = 2/15$ fills the knapsack exactly with part of object 2 and the value of the resulting solution is 28.2. This is solution 2 and it is readily seen to be suboptimal. The method used to obtain this solution is termed a greedy method because at each step (except possibly the last one) we chose to introduce that object which would increase the objective function value the most. However, this greedy method did not yield an optimal solution. Note that even if we change the above strategy so that in the last step the objective function increases by as much as possible, an optimal solution is not obtained for Example 4.1.

We can formulate at least two other greedy approaches attempting to obtain optimal solutions. From the preceding example, we note that considering objects in order of nonincreasing profit values does not yield an optimal solution because even though the objective function value takes on large increases at each step, the number of steps is few as the knapsack capacity is used up at a rapid rate. So, let us try to be greedy with capacity and use it up as slowly as possible. This requires us to consider the objects in order of nondecreasing weights w_i. Using Example 4.1, solution 3 results. This too is suboptimal. This time, even though capacity is used slowly, profits aren't coming in rapidly enough.

Thus, our next attempt is an algorithm that strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used. At each step we include that object which has the maximum profit per unit of capacity used. This means that objects are considered in order of the ratio p_i/w_i. Solution 4 of Example 4.1 is produced by this strategy. If the objects have already been sorted into nonincreasing order of p_i/w_i, then function GreedyKnapsack (Algorithm 4.2) obtains solutions corresponding to this strategy. Note that solutions corresponding to the first two strategies can be obtained using this algorithm if the objects are initially in the appropriate order. Disregarding the time to initially sort the objects, each of the three strategies outlined above requires only $O(n)$ time.

We have seen that when one applies the greedy method to the solution of the knapsack problem, there are at least three different measures one can attempt to optimize when determining which object to include next. These measures are total profit, capacity used, and the ratio of accumulated profit to capacity used. Once an optimization measure has been chosen, the greedy

2. [0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \leq i \leq n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem

$$\max \sum_{1}^{n} p_i x_i$$

$$\text{subject to} \sum_{1}^{n} w_i x_i \leq m$$

$$\text{and} \quad x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

One greedy strategy is to consider the objects in order of nonincreasing density p_i/w_i and add the object into the knapsack if it fits. Show that this strategy doesn't necessarily yield an optimal solution.

4.3 TREE VERTEX SPLITTING

Consider a directed binary tree each edge of which is labeled with a real number (called its *weight*). Trees with edge weights are called *weighted trees*. A weighted tree can be used, for example, to model a distribution network in which electric signals or commodities such as oil are transmitted. Nodes in the tree correspond to receiving stations and edges correspond to transmission lines. It is conceivable that in the process of transmission some loss occurs (drop in voltage in the case of electric signals or drop in pressure in the case of oil). Each edge in the tree is labeled with the loss that occurs in traversing that edge. The network may not be able to tolerate losses beyond a certain level. In places where the loss exceeds the tolerance level, boosters have to be placed. Given a network and a loss tolerance level, the *Tree Vertex Splitting Problem (TVSP)* is to determine an optimal placement of boosters. It is assumed that the boosters can only be placed in the nodes of the tree.

The TVSP can be specified more precisely as follows: Let $T = (V, E, w)$ be a weighted directed tree, where V is the vertex set, E is the edge set, and w is the weight function for the edges. In particular, $w(i, j)$ is the weight of the edge $\langle i, j \rangle \in E$. The weight $w(i, j)$ is undefined for any $\langle i, j \rangle \notin E$. A *source vertex* is a vertex with in-degree zero, and a *sink vertex* is a vertex with out-degree zero. For any path P in the tree, its *delay*, $d(P)$, is defined to be the sum of the weights on that path. The delay of the tree T, $d(T)$, is the maximum of all the path delays.

Let T/X be the forest that results when each vertex u in X is split into two nodes u^i and u^o such that all the edges $\langle u, j \rangle \in E$ ($\langle j, u \rangle \in E$) are

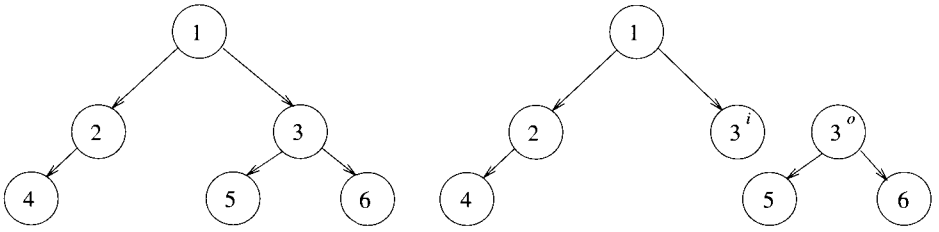


Figure 4.1 A tree before and after splitting the node 3

replaced by edges of the form $\langle u^o, j\rangle$ ($\langle j, u^i\rangle$). In other words, outbound edges from u now leave from u^o and inbound edges to u now enter at u^i. Figure 4.1 shows a tree before and after splitting the node 3. A node that gets split corresponds to a booster station. The TVSP is to identify a set $X \subseteq V$ of minimum cardinality for which $d(T/X) \leq \delta$, for some specified tolerance limit δ. Note that the TVSP has a solution only if the maximum edge weight is $\leq \delta$. Also note that the TVSP naturally fits the subset paradigm.

Given a weighted tree $T(V, E, w)$ and a tolerance limit δ, any subset X of V is a feasible solution if $d(T/X) \leq \delta$. Given an X, we can compute $d(T/X)$ in $O(|V|)$ time. A trivial way of solving the TVSP is to compute $d(T/X)$ for each possible subset X of V. But there are $2^{|V|}$ such subsets! A better algorithm can be obtained using the greedy method.

For the TVSP, the quantity that is optimized (minimized) is the number of nodes in X. A greedy approach to solving this problem is to compute for each node $u \in V$, the maximum delay $d(u)$ from u to any other node in its subtree. If u has a parent v such that $d(u) + w(v, u) > \delta$, then the node u gets split and $d(u)$ is set to zero. Computation proceeds from the leaves toward the root.

In the tree of Figure 4.2, let $\delta = 5$. For each of the leaf nodes $7, 8, 5, 9$, and 10 the delay is zero. The delay for any node is computed only after the delays for its children have been determined. Let u be any node and $C(u)$ be the set of all children of u. Then $d(u)$ is given by

$$d(u) = \max_{v \in C(u)} \{d(v) + w(u, v)\}$$

Using the above formula, for the tree of Figure 4.2, $d(4) = 4$. Since $d(4) + w(2, 4) = 6 > \delta$, node 4 gets split. We set $d(4) = 0$. Now $d(2)$ can be

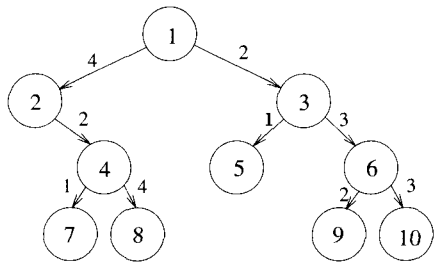


Figure 4.2 An example tree

computed and is equal to 2. Since $d(2) + w(1,2)$ exceeds δ, node 2 gets split and $d(2)$ is set to zero. Then $d(6)$ is equal to 3. Also, since $d(6)+w(3,6) > \delta$, node 6 has to be split. Set $d(6)$ to zero. Now $d(3)$ is computed as 3. Finally, $d(1)$ is computed as 5.

Figure 4.3 shows the final tree that results after splitting the nodes 2, 4, and 6. This algorithm is described in Algorithm 4.3, which is invoked as TVS($root, \delta$), *root* being the root of the tree. The order in which TVS visits (i.e., computes the delay values of) the nodes of the tree is called the *post order* and is studied again in Chapter 6.

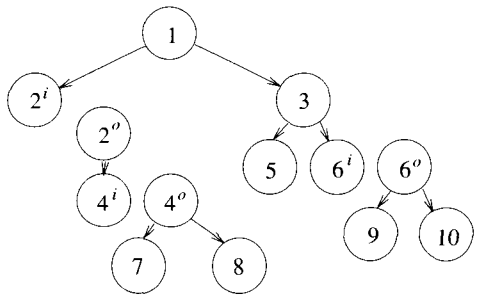


Figure 4.3 The final tree after splitting the nodes 2, 4, and 6

```
1   Algorithm TVS(T, δ)
2   // Determine and output the nodes to be split.
3   // w() is the weighting function for the edges.
4   {
5       if (T ≠ 0) then
6       {
7           d[T] := 0;
8           for each child v of T do
9           {
10              TVS(v, δ);
11              d[T] := max{d[T], d[v] + w(T, v)};
12          }
13          if ((T is not the root)  and
14                  (d[T] + w(parent(T), T) > δ)) then
15          {
16              write (T); d[T] := 0;
17          }
18      }
19  }
```

Algorithm 4.3 The tree vertex splitting algorithm

Algorithm TVS takes $\Theta(n)$ time, where n is the number of nodes in the tree. This can be seen as follows: When TVS is called on any node T, only a constant number of operations are performed (excluding the time taken for the recursive calls). Also, TVS is called only once on each node T in the tree.

Algorithm 4.4 is a revised version of Algorithm 4.3 for the special case of directed binary trees. A sequential representation of the tree (see Section 2.2) has been employed. The tree is stored in the array *tree*[ ] with the root at *tree*[1]. Edge weights are stored in the array *weight*[ ]. If *tree*[*i*] has a tree node, the weight of the incoming edge from its parent is stored in *weight*[*i*]. The delay of node i is stored in $d[i]$. The array $d[\ ]$ is initialized to zero at the beginning. Entries in the arrays *tree*[ ] and *weight*[ ] corresponding to nonexistent nodes will be zero. As an example, for the tree of Figure 4.2, *tree*[ ] will be set to $\{1, 2, 3, 0, 4, 5, 6, 0, 0, 7, 8, 0, 0, 9, 10\}$ starting at cell 1. Also, *weight*[ ] will be set to $\{0, 4, 2, 0, 2, 1, 3, 0, 0, 1, 4, 0, 0, 2, 3\}$ at the beginning, starting from cell 1. The algorithm is invoked as TVS$(1, \delta)$. Now we show that TVS (Algorithm 4.3) will always split a minimal number of nodes.

```
1   Algorithm TVS(i, δ)
2   // Determine and output a minimum cardinality split set.
3   // The tree is realized using the sequential representation.
4   // Root is at tree[1]. N is the largest number such that
5   // tree[N] has a tree node.
6   {
7       if (tree[i] ≠ 0) then // If the tree is not empty
8           if (2i > N) then d[i] := 0; // i is a leaf.
9           else
10          {
11              TVS(2i, δ);
12              d[i] := max(d[i], d[2i] + weight[2i]);
13              if (2i + 1 ≤ N) then
14              {
15                  TVS(2i + 1, δ);
16                  d[i] := max(d[i], d[2i + 1] + weight[2i + 1]);
17              }
18          }
19          if ((tree[i] ≠ 1) and (d[i] + weight[i] > δ)) then
20          {
21              write (tree[i]); d[i] := 0;
22          }
23  }
```

Algorithm 4.4 TVS for the special case of binary trees

Theorem 4.2 Algorithm TVS outputs a minimum cardinality set U such that $d(T/U) \leq \delta$ on any tree T, provided no edge of T has weight $> \delta$.

Proof: The proof is by induction on the number of nodes in the tree. If the tree has a single node, the theorem is true. Assume the theorem for all trees of size $\leq n$. We prove it for trees of size $n+1$ also.

Let T be any tree of size $n+1$ and let U be the set of nodes split by TVS. Also let W be a minimum cardinality set such that $d(T/W) \leq \delta$. We have to show that $|U| \leq |W|$. If $|U| = 0$, this is true. Otherwise, let x be the first vertex split by TVS. Let T_x be the subtree rooted at x. Let T' be the tree obtained from T by deleting T_x except for x. Note that W has to have at least one node, say y, from T_x. Let $W' = W - \{y\}$. If there is a W^* such that $|W^*| < |W'|$ and $d(T'/W^*) \leq \delta$, then since $d(T/(W^* + \{x\})) \leq \delta$, W is not a minimum cardinality split set for T. Thus, W' has to be a minimum cardinality split set such that $d(T'/W') \leq \delta$.

If algorithm TVS is run on tree T', the set of split nodes output is $U-\{x\}$. Since T' has $\leq n$ nodes, $U-\{x\}$ is a minimum cardinality split set for T'. This in turn means that $|W'| \geq |U| - 1$. In other words, $|W| \geq |U|$. □

EXERCISES

1. For the tree of Figure 4.2 solve the TVSP when (a) $\delta = 4$ and (b) $\delta = 6$.

2. Rewrite TVS (Algorithm 4.3) for general trees. Make use of pointers.

4.4 JOB SEQUENCING WITH DEADLINES

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

Example 4.2 Let $n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

To formulate a greedy algorithm to obtain an optimal solution, we must formulate an optimization measure to determine how the next job is chosen. As a first attempt we can choose the objective function $\sum_{i \in J} p_i$ as our optimization measure. Using this measure, the next job to include is the one that increases $\sum_{i \in J} p_i$ the most, subject to the constraint that the resulting J is a feasible solution. This requires us to consider jobs in nonincreasing order of the p_i's. Let us apply this criterion to the data of Example 4.2. We begin with $J = \emptyset$ and $\sum_{i \in J} p_i = 0$. Job 1 is added to J as it has the largest profit and $J = \{1\}$ is a feasible solution. Next, job 4 is considered. The solution $J = \{1, 4\}$ is also feasible. Next, job 3 is considered and discarded as $J = \{1, 3, 4\}$ is not feasible. Finally, job 2 is considered for inclusion into J. It is discarded as $J = \{1, 2, 4\}$ is not feasible. Hence, we are left with the solution $J = \{1, 4\}$ with value 127. This is the optimal solution for the given problem instance. Theorem 4.4 proves that the greedy algorithm just described always obtains an optimal solution to this sequencing problem.

Before attempting the proof, let us see how we can determine whether a given J is a feasible solution. One obvious way is to try out all possible permutations of the jobs in J and check whether the jobs in J can be processed in any one of these permutations (sequences) without violating the deadlines. For a given permutation $\sigma = i_1, i_2, i_3, \ldots, i_k$, this is easy to do, since the earliest time job $i_q, 1 \leq q \leq k$, will be completed is q. If $q > d_{i_q}$, then using σ, at least job i_q will not be completed by its deadline. However, if $|J| = i$, this requires checking $i!$ permutations. Actually, the feasibility of a set J can be determined by checking only one permutation of the jobs in J. This permutation is any one of the permutations in which jobs are ordered in nondecreasing order of deadlines.

Theorem 4.3 Let J be a set of k jobs and $\sigma = i_1, i_2, \ldots, i_k$ a permutation of jobs in J such that $d_{i_1} \leq d_{i_2} \leq \cdots \leq d_{i_k}$. Then J is a feasible solution iff the jobs in J can be processed in the order σ without violating any deadline.

Proof: Clearly, if the jobs in J can be processed in the order σ without violating any deadline, then J is a feasible solution. So, we have only to show that if J is feasible, then σ represents a possible order in which the jobs can be processed. If J is feasible, then there exists $\sigma' = r_1, r_2, \ldots, r_k$ such that $d_{r_q} \geq q$, $1 \leq q \leq k$. Assume $\sigma' \neq \sigma$. Then let a be the least index such that $r_a \neq i_a$. Let $r_b = i_a$. Clearly, $b > a$. In σ' we can interchange r_a and r_b. Since $d_{r_a} \geq d_{r_b}$, the resulting permutation $\sigma'' = s_1, s_2, \ldots, s_k$ represents an order in which the jobs can be processed without violating a deadline. Continuing in this way, σ' can be transformed into σ without violating any deadline. Hence, the theorem is proved. □

Theorem 4.3 is true even if the jobs have different processing times $t_i \geq 0$ (see the exercises).

Theorem 4.4 The greedy method described above always obtains an optimal solution to the job sequencing problem.

Proof: Let $(p_i, d_i), 1 \leq i \leq n$, define any instance of the job sequencing problem. Let I be the set of jobs selected by the greedy method. Let J be the set of jobs in an optimal solution. We now show that both I and J have the same profit values and so I is also optimal. We can assume $I \neq J$ as otherwise we have nothing to prove. Note that if $J \subset I$, then J cannot be optimal. Also, the case $I \subset J$ is ruled out by the greedy method. So, there exist jobs a and b such that $a \in I$, $a \notin J$, $b \in J$, and $b \notin I$. Let a be a highest-profit job such that $a \in I$ and $a \notin J$. It follows from the greedy method that $p_a \geq p_b$ for all jobs b that are in J but not in I. To see this, note that if $p_b > p_a$, then the greedy method would consider job b before job a and include it into I.

Now, consider feasible schedules S_I and S_J for I and J respectively. Let i be a job such that $i \in I$ and $i \in J$. Let i be scheduled from t to $t+1$ in S_I and t' to $t'+1$ in S_J. If $t < t'$, then we can interchange the job (if any) scheduled in $[t', t'+1]$ in S_I with i. If no job is scheduled in $[t', t'+1]$ in I, then i is moved to $[t', t'+1]$. The resulting schedule is also feasible. If $t' < t$, then a similar transformation can be made in S_J. In this way, we can obtain schedules S'_I and S'_J with the property that all jobs common to I and J are scheduled at the same time. Consider the interval $[t_a, t_a + 1]$ in S'_I in which the job a (defined above) is scheduled. Let b be the job (if any) scheduled in S'_J in this interval. From the choice of $a, p_a \geq p_b$. Scheduling a from t_a to $t_a + 1$ in S'_J and discarding job b gives us a feasible schedule for job set $J' = J - \{b\} \cup \{a\}$. Clearly, J' has a profit value no less than that of J and differs from I in one less job than J does.

By repeatedly using the transformation just described, J can be transformed into I with no decrease in profit value. So I must be optimal. □

A high-level description of the greedy algorithm just discussed appears as Algorithm 4.5. This algorithm constructs an optimal set J of jobs that can be processed by their due times. The selected jobs can be processed in the order given by Theorem 4.3.

Now, let us see how to represent the set J and how to carry out the test of lines 7 and 8 in Algorithm 4.5. Theorem 4.3 tells us how to determine whether all jobs in $J \cup \{i\}$ can be completed by their deadlines. We can avoid sorting the jobs in J each time by keeping the jobs in J ordered by deadlines. We can use an array $d[1:n]$ to store the deadlines of the jobs in the order of their p-values. The set J itself can be represented by a one-dimensional array $J[1:k]$ such that $J[r]$, $1 \leq r \leq k$ are the jobs in J and $d[J[1]] \leq d[J[2]] \leq \cdots \leq d[J[k]]$. To test whether $J \cup \{i\}$ is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d[J[r]] \leq r$, $1 \leq r \leq k+1$. The insertion of i into J is simplified by the use of a fictitious job 0 with $d[0] = 0$ and $J[0] = 0$. Note also that if job i is to be inserted at position q, then only the positions of jobs $J[q]$, $J[q+1]$,

```
1   Algorithm GreedyJob(d, J, n)
2   // J is a set of jobs that can be completed by their deadlines.
3   {
4       J := {1};
5       for i := 2 to n do
6       {
7           if (all jobs in J ∪ {i} can be completed
8               by their deadlines) then J := J ∪ {i};
9       }
10  }
```

Algorithm 4.5 High-level description of job sequencing algorithm

$\ldots$, $J[k]$ are changed after the insertion. Hence, it is necessary to verify only that these jobs (and also job i) do not violate their deadlines following the insertion. The algorithm that results from this discussion is function JS (Algorithm 4.6). The algorithm assumes that the jobs are already sorted such that $p_1 \geq p_2 \geq \cdots \geq p_n$. Further it assumes that $n \geq 1$ and the deadline $d[i]$ of job i is at least 1. Note that no job with $d[i] < 1$ can ever be finished by its deadline. Theorem 4.5 proves that JS is a correct implementation of the greedy strategy.

Theorem 4.5 Function JS is a correct implementation of the greedy-based method described above.

Proof: Since $d[i] \geq 1$, the job with the largest p_i will always be in the greedy solution. As the jobs are in nonincreasing order of the p_i's, line 8 in Algorithm 4.6 includes the job with largest p_i. The **for** loop of line 10 considers the remaining jobs in the order required by the greedy method described earlier. At all times, the set of jobs already included in the solution is maintained in J. If $J[i]$, $1 \leq i \leq k$, is the set already included, then J is such that $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$. This allows for easy application of the feasibility test of Theorem 4.3. When job i is being considered, the **while** loop of line 15 determines where in J this job has to be inserted. The use of a fictitious job 0 (line 7) allows easy insertion into position 1. Let w be such that $d[J[w]] \leq d[i]$ and $d[J[q]] > d[i]$, $w < q \leq k$. If job i is included into J, then jobs $J[q]$, $w < q \leq k$, have to be moved one position up in J (line 19). From Theorem 4.3, it follows that such a move retains feasibility of J iff $d[J[q]] \neq q$, $w < q \leq k$. This condition is verified in line 15. In addition, i can be inserted at position $w+1$ iff $d[i] > w$. This is verified in line 16 (note $r = w$ on exit from the **while** loop if $d[J[q]] \neq q$, $w < q \leq k$). The correctness of JS follows from these observations. □

```
1   Algorithm JS(d, j, n)
2   // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3   // are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
4   // is the ith job in the optimal solution, 1 ≤ i ≤ k.
5   // Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
6   {
7       d[0] := J[0] := 0; // Initialize.
8       J[1] := 1; // Include job 1.
9       k := 1;
10      for i := 2 to n do
11      {
12          // Consider jobs in nonincreasing order of p[i]. Find
13          // position for i and check feasibility of insertion.
14          r := k;
15          while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
16          if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17          {
18              // Insert i into J[ ].
19              for q := k to (r + 1) step −1 do J[q + 1] := J[q];
20              J[r + 1] := i; k := k + 1;
21          }
22      }
23      return k;
24  }
```

Algorithm 4.6 Greedy algorithm for sequencing unit time jobs with deadlines and profits

For JS there are two possible parameters in terms of which its complexity can be measured. We can use n, the number of jobs, and s, the number of jobs included in the solution J. The **while** loop of line 15 in Algorithm 4.6 is iterated at most k times. Each iteration takes $\Theta(1)$ time. If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require $\Theta(k - r)$ time to insert job i. Hence, the total time for each iteration of the **for** loop of line 10 is $\Theta(k)$. This loop is iterated $n - 1$ times. If s is the final value of k, that is, s is the number of jobs in the final solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \leq n$, the worst-case time, as a function of n alone is $\Theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1$, $1 \leq i \leq n$, then algorithm JS takes $\Theta(n^2)$ time to determine J. Hence, the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space needed for d, JS needs $\Theta(s)$ amount of space for J.

Note that the profit values are not needed by JS. It is sufficient to know that $p_i \geq p_{i+1}$, $1 \leq i < n$.

The computing time of JS can be reduced from $O(n^2)$ to nearly $O(n)$ by using the disjoint set union and find algorithms (see Section 2.5) and a different method to determine the feasibility of a partial solution. If J is a feasible subset of jobs, then we can determine the processing times for each of the jobs using the rule: if job i hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the largest integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free. This rule simply delays the processing of job i as much as possible. Consequently, when J is being built up job by job, jobs already in J do not have to be moved from their assigned slots to accommodate the new job. If for the new job being considered there is no α as defined above, then it cannot be included in J. The proof of the validity of this statement is left as an exercise.

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

Since there are only n jobs and each job takes one unit of time, it is necessary only to consider the time slots $[i - 1, i]$, $1 \leq i \leq b$, such that $b = \min\{n, \max\{d_i\}\}$. One way to implement the above scheduling rule is to partition the time slots $[i-1, i]$, $1 \leq i \leq b$, into sets. We use i to represent the time slots $[i - 1, i]$. For any slot i, let n_i be the largest integer such that $n_i \leq i$ and slot n_i is free. To avoid end conditions, we introduce a fictitious slot $[-1, 0]$ which is always free. Two slots i and j are in the same set iff $n_i = n_j$. Clearly, if i and j, $i < j$, are in the same set, then $i, i+1, i+2, \ldots, j$ are in the same set. Associated with each set k of slots is a value $f(k)$. Then $f(k) = n_i$ for all slots i in set k. Using the set representation of Section 2.5, each set is represented as a tree. The root node identifies the set. The function f is defined only for root nodes. Initially, all slots are free and we have $b + 1$ sets corresponding to the $b + 1$ slots $[i - 1, i]$, $0 \leq i \leq b$. At this time $f(i) = i$, $0 \leq i \leq b$. We use $p(i)$ to link slot i into its set tree. With the conventions for the union and find algorithms of Section 2.5, $p(i) = -1$, $0 \leq i \leq b$, initially. If a job with deadline d is to be scheduled, then we need to find the root of the tree containing the slot $\min\{n, d\}$. If this root is j,

then $f(j)$ is the nearest free slot, provided $f(j) \neq 0$. Having used this slot, the set with root j should be combined with the set containing slot $f(j)-1$.

Example 4.4 The trees defined by the $p(i)$'s for the first three iterations in Example 4.3 are shown in Figure 4.4. □

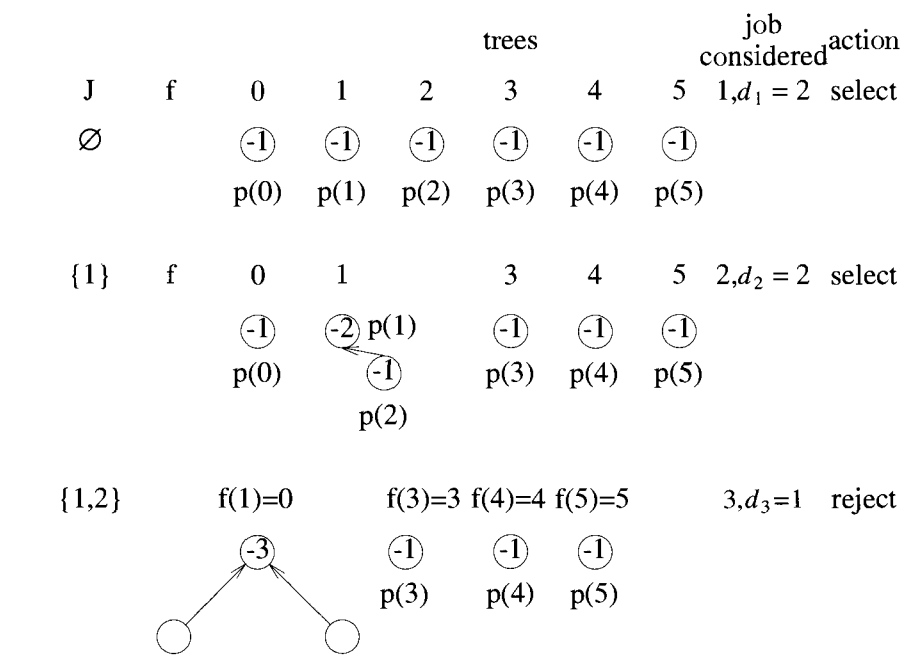


Figure 4.4 Fast job scheduling

The fast algorithm appears as FJS (Algorithm 4.7). Its computing time is readily observed to be $O(n\alpha(2n,n))$ (recall that $\alpha(2n,n)$ is the inverse of Ackermann's function defined in Section 2.5). It needs an additional $2n$ words of space for f and p.

```
1   Algorithm FJS(d, n, b, j)
2   // Find an optimal solution J[1 : k]. It is assumed that
3   // p[1] ≥ p[2] ≥ ··· ≥ p[n] and that b = min{n, max_i(d[i])}.
4   {
5       // Initially there are b + 1 single node trees.
6       for i := 0 to b do f[i] := i;
7       k := 0; // Initialize.
8       for i := 1 to n do
9       { // Use greedy rule.
10          q := CollapsingFind(min(n, d[i]));
11          if (f[q] ≠ 0) then
12          {
13              k := k + 1; J[k] := i; // Select job i.
14              m := CollapsingFind(f[q] − 1);
15              WeightedUnion(m, q);
16              f[q] := f[m]; // q may be new root.
17          }
18      }
19  }
```

Algorithm 4.7 Faster algorithm for job sequencing

EXERCISES

1. You are given a set of n jobs. Associated with each job i is a processing time t_i and a deadline d_i by which it must be completed. A feasible schedule is a permutation of the jobs such that if the jobs are processed in that order, then each job finishes by its deadline. Define a greedy schedule to be one in which the jobs are processed in nondecreasing order of deadlines. Show that if there exists a feasible schedule, then all greedy schedules are feasible.

2. [Optimal assignment] Assume there are n workers and n jobs. Let v_{ij} be the value of assigning worker i to job j. An assignment of workers to jobs corresponds to the assignment of 0 or 1 to the variables x_{ij}, $1 \leq i, j \leq n$. Then $x_{ij} = 1$ means worker i is assigned to job j, and $x_{ij} = 0$ means that worker i is not assigned to job j. A valid assignment is one in which each worker is assigned to exactly one job and exactly one worker is assigned to any one job. The value of an assignment is $\sum_i \sum_j v_{ij} x_{ij}$.

For example, assume there are three workers w_1, w_2, and w_3 and three jobs j_1, j_2, and j_3. Let the values of assignment be $v_{11} = 11$, $v_{12} = 5$, $v_{13} = 8$, $v_{21} = 3$, $v_{22} = 7$, $v_{23} = 15$, $v_{31} = 8$, $v_{32} = 12$, and $v_{33} = 9$. Then, a valid assignment is $x_{12} = 1$, $x_{23} = 1$, and $x_{31} = 1$. The rest of the x_{ij}'s are zeros. The value of this assignment is $5 + 15 + 8 = 28$.

An optimal assignment is a valid assignment of maximum value. Write algorithms for two different greedy assignment schemes. One of these assigns a worker to the best possible job. The other assigns to a job the best possible worker. Show that neither of these schemes is guaranteed to yield optimal assignments. Is either scheme always better than the other? Assume $v_{ij} > 0$.

3. (a) What is the solution generated by the function JS when $n = 7$, $(p_1, p_2, \ldots, p_7) = (3, 5, 20, 18, 1, 6, 30)$, and $(d_1, d_2, \ldots, d_7) = (1, 3, 4, 3, 2, 1, 2)$?

   (b) Show that Theorem 4.3 is true even if jobs have different processing requirements. Associated with job i is a profit $p_i > 0$, a time requirement $t_i > 0$, and a deadline $d_i \geq t_i$.

   (c) Show that for the situation of part (a), the greedy method of this section doesn't necessarily yield an optimal solution.

4. (a) For the job sequencing problem of this section, show that the subset J represents a feasible solution iff the jobs in J can be processed according to the rule: if job i in J hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the least integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free.

   (b) For the problem instance of Exercise 3(a) draw the trees and give the values of $f(i), 0 \leq i \leq n$, after each iteration of the **for** loop of line 8 of Algorithm 4.7.

4.5 MINIMUM-COST SPANNING TREES

Definition 4.1 Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of G is a *spanning tree* of G iff t is a tree. □

Example 4.5 Figure 4.5 shows the complete graph on four nodes together with three of its spanning trees. □

Spanning trees have many applications. For example, they can be used to obtain an independent set of circuit equations for an electric network. First, a spanning tree for the electric network is obtained. Let B be the set of network edges not in the spanning tree. Adding an edge from B to

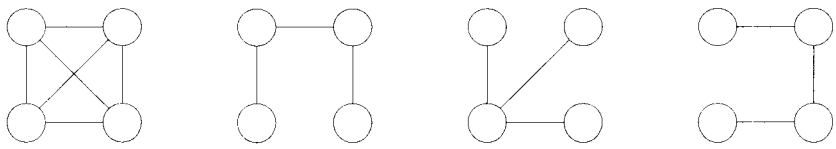


Figure 4.5 An undirected graph and three of its spanning trees

the spanning tree creates a cycle. Kirchoff's second law is used on each cycle to obtain a circuit equation. The cycles obtained in this way are independent (i.e., none of these cycles can be obtained by taking a linear combination of the remaining cycles) as each contains an edge from B that is not contained in any other cycle. Hence, the circuit equations so obtained are also independent. In fact, it can be shown that the cycles obtained by introducing the edges of B one at a time into the resulting spanning tree form a cycle basis, and so all other cycles in the graph can be constructed by taking a linear combination of the cycles in the basis.

Another application of spanning trees arises from the property that a spanning tree is a minimal subgraph G' of G such that $V(G') = V(G)$ and G' is connected. (A minimal subgraph is one with the fewest number of edges.) Any connected graph with n vertices must have at least $n-1$ edges and all connected graphs with $n-1$ edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is $n-1$. The spanning trees of G represent all feasible choices.

In practical situations, the edges have weights assigned to them. These weights may represent the cost of construction, the length of the link, and so on. Given such a weighted graph, one would then wish to select cities to have minimum total cost or minimum total length. In either case the links selected have to form a tree (assuming all weights are positive). If this is not so, then the selection of links contains a cycle. Removal of any one of the links on this cycle results in a link selection of less cost connecting all cities. We are therefore interested in finding a spanning tree of G with minimum cost. (The cost of a spanning tree is the sum of the costs of the edges in that tree.) Figure 4.6 shows a graph and one of its minimum-cost spanning trees. Since the identification of a minimum-cost spanning tree involves the selection of a subset of the edges, this problem fits the subset paradigm.

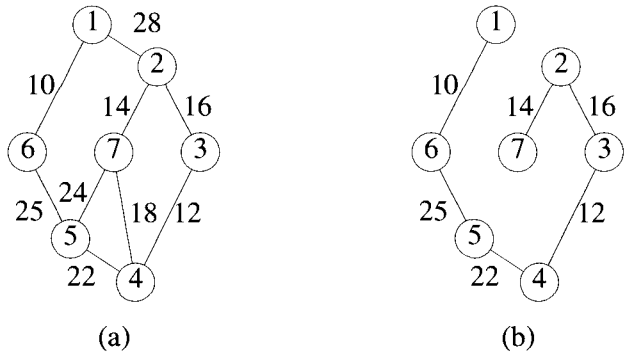


Figure 4.6 A graph and its minimum cost spanning tree

4.5.1 Prim's Algorithm

A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge. The next edge to include is chosen according to some optimization criterion. The simplest such criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included. There are two possible ways to interpret this criterion. In the first, the set of edges so far selected form a tree. Thus, if A is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be included in A is a minimum-cost edge not in A with the property that $A \cup \{(u, v)\}$ is also a tree. Exercise 2 shows that this selection criterion results in a minimum-cost spanning tree. The corresponding algorithm is known as Prim's algorithm.

Example 4.6 Figure 4.7 shows the working of Prim's method on the graph of Figure 4.6(a). The spanning tree obtained is shown in Figure 4.6(b) and has a cost of 99. □

Having seen how Prim's method works, let us obtain a pseudocode algorithm to find a minimum-cost spanning tree using this method. The algorithm will start with a tree that includes only a minimum-cost edge of G. Then, edges are added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included, and the cost of (i, j), $cost[i, j]$, is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree. To determine this edge (i, j) efficiently, we associate with each vertex j not yet included in the tree a value $near[j]$. The value $near[j]$ is a vertex in the tree such that $cost[j, near[j]]$ is minimum among all choices for $near[j]$. We define $near[j] = 0$ for all vertices j that are already in the tree. The next edge

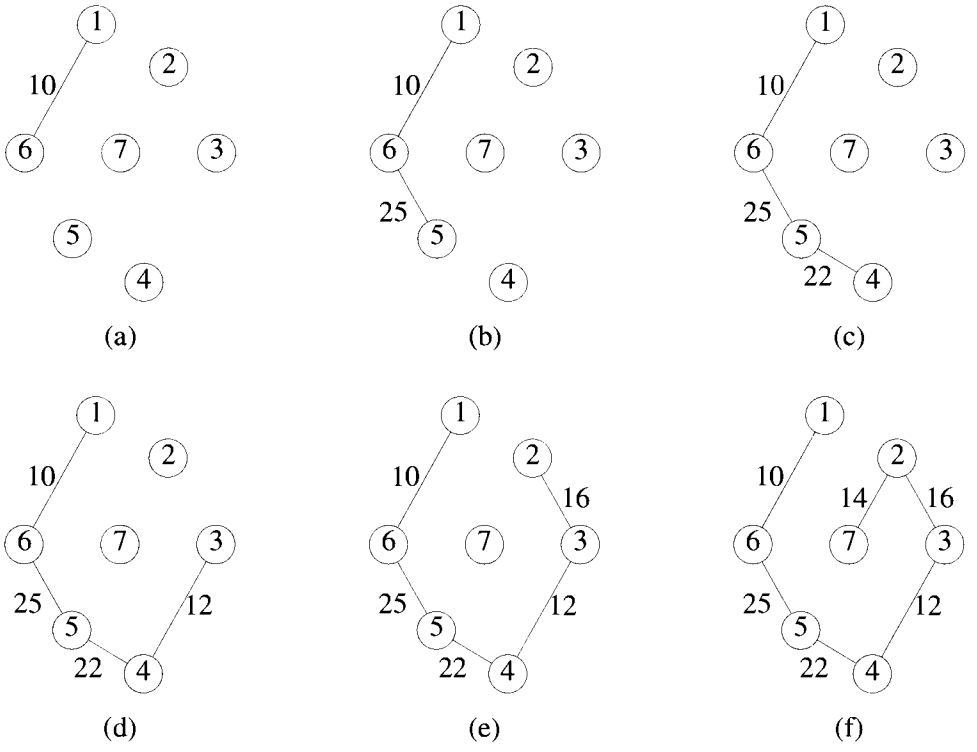


Figure 4.7 Stages in Prim's algorithm

to include is defined by the vertex j such that $near[j] \neq 0$ (j not already in the tree) and $cost[j, near[j]]$ is minimum.

In function Prim (Algorithm 4.8), line 9 selects a minimum-cost edge. Lines 10 to 15 initialize the variables so as to represent a tree comprising only the edge (k, l). In the **for** loop of line 16 the remainder of the spanning tree is built up edge by edge. Lines 18 and 19 select $(j, near[j])$ as the next edge to include. Lines 23 to 25 update $near[\,]$.

The time required by algorithm Prim is $O(n^2)$, where n is the number of vertices in the graph G. To see this, note that line 9 takes $O(|E|)$ time and line 10 takes $\Theta(1)$ time. The **for** loop of line 12 takes $\Theta(n)$ time. Lines 18 and 19 and the **for** loop of line 23 require $O(n)$ time. So, each iteration of the **for** loop of line 16 takes $O(n)$ time. The total time for the **for** loop of line 16 is therefore $O(n^2)$. Hence, Prim runs in $O(n^2)$ time.

If we store the nodes not yet included in the tree as a red-black tree (see Section 2.4.2), lines 18 and 19 take $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). The **for** loop of line 23 has to examine only the nodes adjacent to j. Thus its overall frequency is $O(|E|)$. Updating in lines 24 and 25 also takes $O(\log n)$ time (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O((n + |E|) \log n)$.

The algorithm can be speeded a bit by making the observation that a minimum-cost spanning tree includes for each vertex v a minimum-cost edge incident to v. To see this, suppose t is a minimum-cost spanning tree for $G = (V, E)$. Let v be any vertex in t. Let (v, w) be an edge with minimum cost among all edges incident to v. Assume that $(v, w) \notin E(t)$ and $cost[v, w] < cost[v, x]$ for all edges $(v, x) \in E(t)$. The inclusion of (v, w) into t creates a unique cycle. This cycle must include an edge (v, x), $x \neq w$. Removing (v, x) from $E(t) \cup \{(v, w)\}$ breaks this cycle without disconnecting the graph $(V, E(t) \cup \{(v, w)\})$. Hence, $(V, E(t) \cup \{(v, w)\} - \{(v, x)\})$ is also a spanning tree. Since $cost[v, w] < cost[v, x]$, this spanning tree has lower cost than t. This contradicts the assumption that t is a minimum-cost spanning tree of G. So, t includes minimum-cost edges as stated above.

From this observation it follows that we can start the algorithm with a tree consisting of any arbitrary vertex and no edge. Then edges can be added one by one. The changes needed are to lines 9 to 17. These lines can be replaced by the lines

```
9'        mincost := 0;
10'       for i := 2 to n do near[i] := 1;
11'           // Vertex 1 is initially in t.
12'       near[1] := 0;
13'-16'   for i := 1 to n - 1 do
17'       { // Find n - 1 edges for t.
```

4.5.2 Kruskal's Algorithm

There is a second possible interpretation of the optimization criteria mentioned earlier in which the edges of the graph are considered in nondecreasing order of cost. This interpretation is that the set t of edges so far selected for the spanning tree be such that it is possible to *complete* t into a tree. Thus t may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges t can be completed into a tree iff there are no cycles in t. We show in Theorem 4.6 that this interpretation of the greedy method also results in a minimum-cost spanning tree. This method is due to Kruskal.

```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

Algorithm 4.8 Prim's minimum-cost spanning tree algorithm

Example 4.7 Consider the graph of Figure 4.6(a). We begin with no edges selected. Figure 4.8(a) shows the current graph with no edges selected. Edge $(1,6)$ is the first edge considered. It is included in the spanning tree being built. This yields the graph of Figure 4.8(b). Next, the edge $(3,4)$ is selected and included in the tree (Figure 4.8(c)). The next edge to be considered is $(2,7)$. Its inclusion in the tree being built does not create a cycle, so we get the graph of Figure 4.8(d). Edge $(2,3)$ is considered next and included in the tree Figure 4.8(e). Of the edges not yet considered, $(7,4)$ has the least cost. It is considered next. Its inclusion in the tree results in a cycle, so this edge is discarded. Edge $(5,4)$ is the next edge to be added to the tree being built. This results in the configuration of Figure 4.8(f). The next edge to be considered is the edge $(7,5)$. It is discarded, as its inclusion creates a cycle. Finally, edge $(6,5)$ is considered and included in the tree being built. This completes the spanning tree. The resulting tree (Figure 4.6(b)) has cost 99. □

For clarity, Kruskal's method is written out more formally in Algorithm 4.9. Initially E is the set of all edges in G. The only functions we wish to perform on this set are (1) determine an edge with minimum cost (line 4) and (2) delete this edge (line 5). Both these functions can be performed efficiently if the edges in E are maintained as a sorted sequential list. It is not essential to sort all the edges so long as the next edge for line 4 can be determined easily. If the edges are maintained as a minheap, then the next edge to consider can be obtained in $O(\log |E|)$ time. The construction of the heap itself takes $O(|E|)$ time.

To be able to perform step 6 efficiently, the vertices in G should be grouped together in such a way that one can easily determine whether the vertices v and w are already connected by the earlier selection of edges. If they are, then the edge (v,w) is to be discarded. If they are not, then (v,w) is to be added to t. One possible grouping is to place all vertices in the same connected component of t into a set (all connected components of t will also be trees). Then, two vertices v and w are connected in t iff they are in the same set. For example, when the edge $(2,6)$ is to be considered, the sets are $\{1,2\}, \{3,4,6\}$, and $\{5\}$. Vertices 2 and 6 are in different sets so these sets are combined to give $\{1,2,3,4,6\}$ and $\{5\}$. The next edge to be considered is $(1,4)$. Since vertices 1 and 4 are in the same set, the edge is rejected. The edge $(3,5)$ connects vertices in different sets and results in the final spanning tree. Using the set representation and the union and find algorithms of Section 2.5, we can obtain an efficient (almost linear) implementation of line 6. The computing time is, therefore, determined by the time for lines 4 and 5, which in the worst case is $O(|E| \log |E|)$.

If the representations discussed above are used, then the pseudocode of Algorithm 4.10 results. In line 6 an initial heap of edges is constructed. In line 7 each vertex is assigned to a distinct set (and hence to a distinct tree). The set t is the set of edges to be included in the minimum-cost spanning

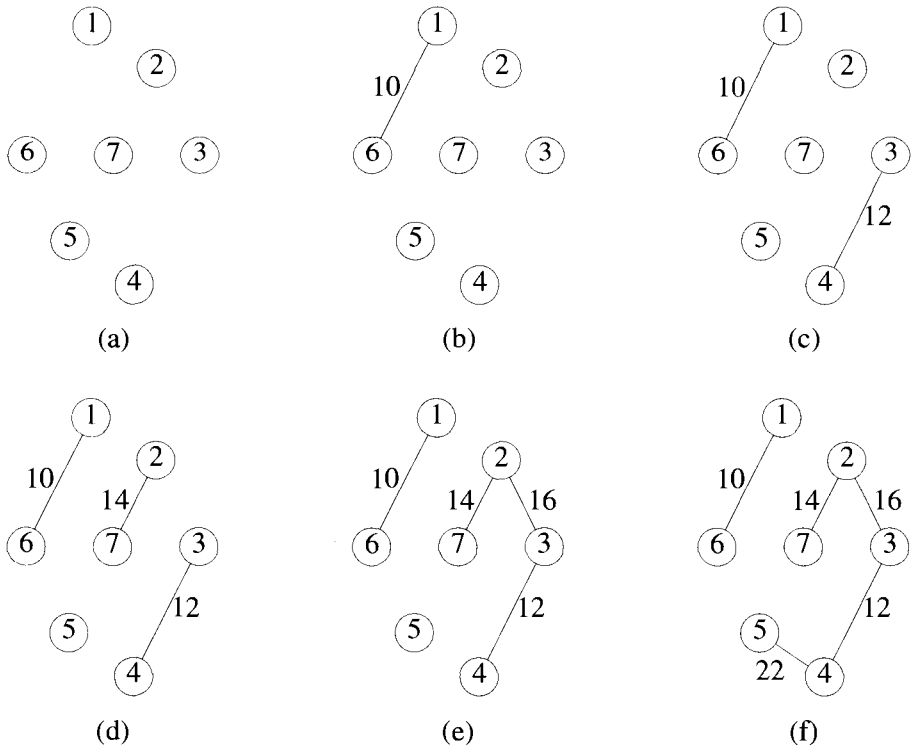


Figure 4.8 Stages in Kruskal's algorithm

tree and i is the number of edges in t. The set t can be represented as a sequential list using a two-dimensional array $t[1:n-1, 1:2]$. Edge (u, v) can be added to t by the assignments $t[i, 1] := u$; and $t[i, 2] := v$;. In the **while** loop of line 10, edges are removed from the heap one by one in nondecreasing order of cost. Line 14 determines the sets containing u and v. If $j \neq k$, then vertices u and v are in different sets (and so in different trees) and edge (u, v) is included into t. The sets containing u and v are combined (line 20). If $u = v$, the edge (u, v) is discarded as its inclusion into t would create a cycle. Line 23 determines whether a spanning tree was found. It follows that $i \neq n - 1$ iff the graph G is not connected. One can verify that the computing time is $O(|E| \log |E|)$, where E is the edge set of G.

Theorem 4.6 Kruskal's algorithm generates a minimum-cost spanning tree for every connected undirected graph G.

```
1  t := ∅;
2  while ((t has less than n − 1 edges) and (E ≠ ∅)) do
3  {
4      Choose an edge (v, w) from E of lowest cost;
5      Delete (v, w) from E;
6      if (v, w) does not create a cycle in t then add (v, w) to t;
7      else discard (v, w);
8  }
```

Algorithm 4.9 Early form of minimum-cost spanning tree algorithm due to Kruskal

```
1  Algorithm Kruskal(E, cost, n, t)
2  // E is the set of edges in G. G has n vertices. cost[u, v] is the
3  // cost of edge (u, v). t is the set of edges in the minimum-cost
4  // spanning tree. The final cost is returned.
5  {
6      Construct a heap out of the edge costs using Heapify;
7      for i := 1 to n do parent[i] := −1;
8      // Each vertex is in a different set.
9      i := 0; mincost := 0.0;
10     while ((i < n − 1) and (heap not empty)) do
11     {
12         Delete a minimum cost edge (u, v) from the heap
13         and reheapify using Adjust;
14         j := Find(u); k := Find(v);
15         if (j ≠ k) then
16         {
17             i := i + 1;
18             t[i, 1] := u; t[i, 2] := v;
19             mincost := mincost + cost[u, v];
20             Union(j, k);
21         }
22     }
23     if (i ≠ n − 1) then write ("No spanning tree");
24     else return mincost;
25 }
```

Algorithm 4.10 Kruskal's algorithm

Proof: Let G be any undirected connected graph. Let t be the spanning tree for G generated by Kruskal's algorithm. Let t' be a minimum-cost spanning tree for G. We show that both t and t' have the same cost.

Let $E(t)$ and $E(t')$ respectively be the edges in t and t'. If n is the number of vertices in G, then both t and t' have $n-1$ edges. If $E(t) = E(t')$, then t is clearly of minimum cost. If $E(t) \neq E(t')$, then let q be a minimum-cost edge such that $q \in E(t)$ and $q \notin E(t')$. Clearly, such a q must exist. The inclusion of q into t' creates a unique cycle (Exercise 5). Let $q, e_1, e_2, \ldots, e_k$ be this unique cycle. At least one of the e_i's, $1 \leq i \leq k$, is not in $E(t)$ as otherwise t would also contain the cycle $q, e_1, e_2, \ldots, e_k$. Let e_j be an edge on this cycle such that $e_j \notin E(t)$. If e_j is of lower cost than q, then Kruskal's algorithm will consider e_j before q and include e_j into t. To see this, note that all edges in $E(t)$ of cost less than the cost of q are also in $E(t')$ and do not form a cycle with e_j. So $cost(e_j) \geq cost(q)$.

Now, reconsider the graph with edge set $E(t') \cup \{q\}$. Removal of any edge on the cycle $q, e_1, e_2, \ldots, e_k$ will leave behind a tree t'' (Exercise 5). In particular, if we delete the edge e_j, then the resulting tree t'' will have a cost no more than the cost of t' (as $cost(e_j) \geq cost(e)$). Hence, t'' is also a minimum-cost tree.

By repeatedly using the transformation described above, tree t' can be transformed into the spanning tree t without any increase in cost. Hence, t is a minimum-cost spanning tree. □

4.5.3 An Optimal Randomized Algorithm (*)

Any algorithm for finding the minimum-cost spanning tree of a given graph $G(V, E)$ will have to spend $\Omega(|V| + |E|)$ time in the worst case, since it has to examine each node and each edge at least once before determining the correct answer. A randomized Las Vegas algorithm that runs in time $\tilde{O}(|V| + |E|)$ can be devised as follows: (1) Randomly sample m edges from G (for some suitable m). (2) Let G' be the induced subgraph; that is, G' has V as its node set and the sampled edges in its edge set. The subgraph G' need not be connected. Recursively find a minimum-cost spanning tree for each component of G'. Let F be the resultant *minimum-cost spanning forest* of G'. (3) Using F, eliminate certain edges (called the *F-heavy edges*) of G that cannot possibly be in a minimum-cost spanning tree. Let G'' be the graph that results from G after elimination of the F-heavy edges. (4) Recursively find a minimum-cost spanning tree for G''. This will also be a minimum-cost spanning tree for G.

Steps 1 to 3 are useful in reducing the number of edges in G. The algorithm can be speeded up further if we can reduce the number of nodes in the input graph as well. Such a node elimination can be effected using the *Borůvka steps*. In a Borůvka step, for each node, an incident edge with minimum weight is chosen. For example in Figure 4.9(a), the edge $(1, 3)$ is

chosen for node 1, the edge $(6,7)$ is chosen for node 7, and so on. All the chosen edges are shown with thick lines. The connected components of the induced graph are found. In the example of Figure 4.9(a), the nodes 1, 2, and 3 form one component, the nodes 4 and 5 form a second component, and the nodes 6 and 7 form another component. Replace each component with a single node. The component with nodes 1, 2, and 3 is replaced with the node a. The other two components are replaced with the nodes b and c, respectively. Edges within the individual components are thrown away. The resultant graph is shown in Figure 4.9(b). In this graph keep only an edge of minimum weight between any two nodes. Delete any isolated nodes.

Since an edge is chosen for every node, the number of nodes after one Borůvka step reduces by a factor of at least two. A minimum-cost spanning tree for the reduced graph can be extended easily to get a minimum-cost spanning tree for the original graph. If E' is the set of edges in the minimum-cost spanning tree of the reduced graph, we simply include into E' the edges chosen in the Borůvka step to obtain the minimum-cost spanning tree edges for the original graph. In the example of Figure 4.9, a minimum-cost spanning tree for (c) will consist of the edges (a,b) and (b,c). Thus a minimum-cost spanning tree for the graph of (a) will have the edges: $(1,3),(3,2),(4,5),(6,7),(3,4)$, and $(2,6)$. More details of the algorithms are given below.

Definition 4.2 Let F be a forest that forms a subgraph of a given weighted graph $G(V,E)$. If u and v are any two nodes in F, let $F(u,v)$ denote the path (if any) connecting u and v in F and let $Fcost(u,v)$ denote the maximum weight of any edge in the path $F(u,v)$. If there is no path between u and v in F, $Fcost(u,v)$ is taken to be ∞. Any edge (x,y) of G is said to be F-heavy if $cost[x,y] > Fcost(x,y)$ and F-light otherwise. □

Note that all the edges of F are F-light. Also, any F-heavy edge cannot belong to a minimum-cost spanning tree of G. The proof of this is left as an exercise. The randomized algorithm applies two Borůvka steps to reduce the number of nodes in the input graph. Next, it samples the edges of G and processes them to eliminate a constant fraction of them. A minimum-cost spanning tree for the resultant reduced graph is recursively computed. From this tree, a spanning tree for G is obtained. A detailed description of the algorithm appears as Algorithm 4.11.

Lemma 4.3 states that Step 4 can be completed in time $O(|V|+|E|)$. The proof of this can be found in the references supplied at the end of this chapter. Step 1 takes $O(|V|+|E|)$ time and step 2 takes $O(|E|)$ time. Step 6 takes $O(|E|)$ time as well. The time taken in all the recursive calls in steps 3 and 5 can be shown to be $O(|V|+|E|)$. For a proof, see the references at the end of the chapter. A crucial fact that is used in the proof is that both the number of nodes and the number of edges are reduced by a constant factor, with high probability, in each level of recursion.

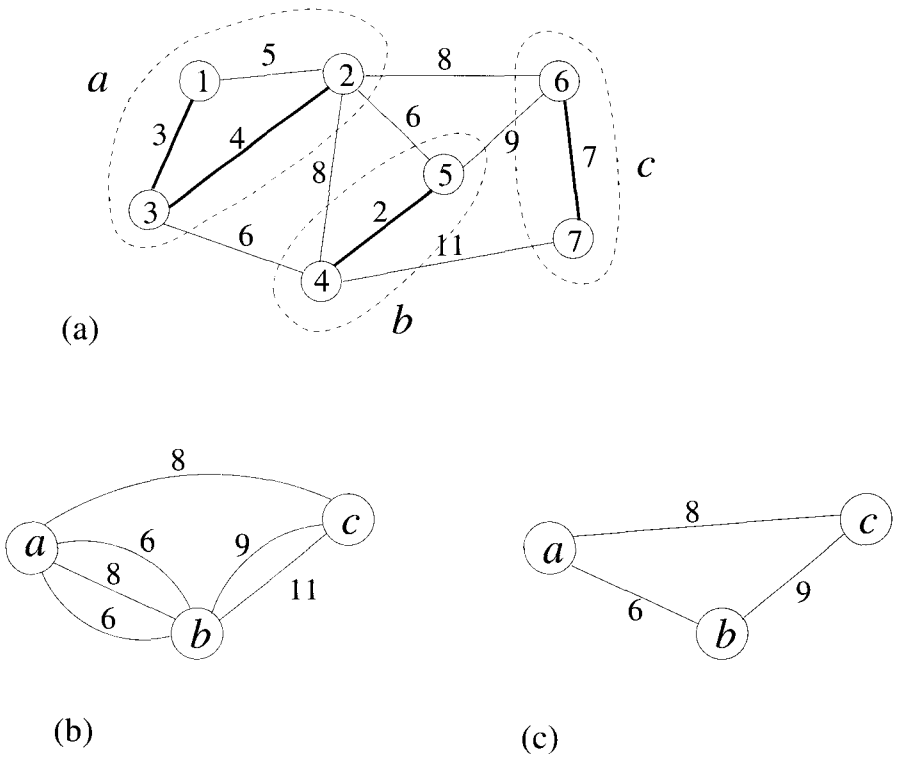


Figure 4.9 A Borůvka step

Lemma 4.3 Let $G(V, E)$ be any weighted graph and let F be a subgraph of G that forms a forest. Then, all the F-heavy edges of G can be identified in time $O(|V| + |E|)$. □

Theorem 4.7 A minimum-weight spanning tree for any given weighted graph can be computed in time $\tilde{O}(|V| + |E|)$. □

EXERCISES

1. Compute a minimum cost spanning tree for the graph of Figure 4.10 using (a) Prim's algorithm and (b) Kruskal's algorithm.

2. Prove that Prim's method of this section generates minimum-cost spanning trees.

Step 1. Apply two Borůvka steps. At the end, the number of nodes will have decreased by a factor at least 4. Let the resultant graph be $\widetilde{G}(\widetilde{V}, \widetilde{E})$.

Step 2. Form a subgraph $G'(V', E')$ of $\widetilde{G}$, where each edge of $\widetilde{G}$ is chosen randomly to be in E' with probability $\frac{1}{2}$. The expected number of edges in E' is $\frac{|\widetilde{E}|}{2}$.

Step 3. Recursively find a minimum-cost spanning forest F for G'.

Step 4. Eliminate all the F-heavy edges from $\widetilde{G}$. With high probability, at least a constant fraction of the edges of $\widetilde{G}$ will be eliminated. Let G'' be the resultant graph.

Step 5. Compute a minimum-cost spanning tree (call it T'') for G'' recursively. The tree T'' will also be a minimum-cost spanning tree for $\widetilde{G}$.

Step 6. Return the edges of T'' together with the edges chosen in the Borůvka steps of step 1. These are the edges of a minimum-cost spanning tree for G.

Algorithm 4.11 An optimal randomized algorithm

3. (a) Rewrite Prim's algorithm under the assumption that the graphs are represented by adjacency lists.
   (b) Program and run the above version of Prim's algorithm against Algorithm 4.9. Compare the two on a representative set of graphs.
   (c) Analyze precisely the computing time and space requirements of your new version of Prim's algorithm using adjacency lists.

4. Program and run Kruskal's algorithm, described in Algorithm 4.10. You will have to modify functions Heapify and Adjust of Chapter 2. Use the same test data you devised to test Prim's algorithm in Exercise 3.

5. (a) Show that if t is a spanning tree for the undirected graph G, then the addition of an edge q, $q \notin E(t)$ and $q \in E(G)$, to t creates a unique cycle.

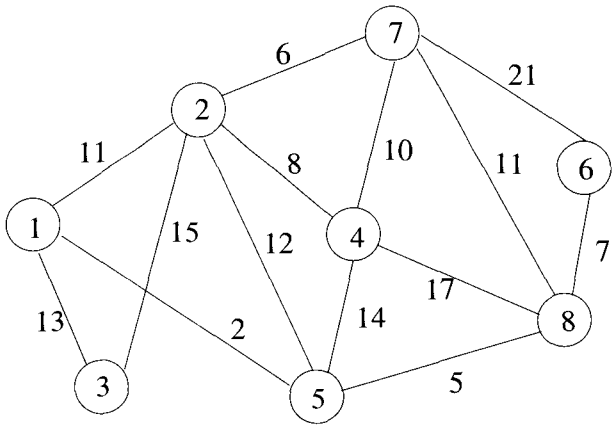


Figure 4.10 Graph for Exercise 1

(b) Show that if any of the edges on this unique cycle is deleted from $E(t) \cup \{q\}$, then the remaining edges form a spanning tree of G.

6. In Figure 4.9, find a minimum-cost spanning tree for the graph of part (c) and extend the tree to obtain a minimum cost spanning tree for the graph of part (a). Verify the correctness of your answer by applying either Prim's algorithm or Kruskal's algorithm on the graph of part (a).

7. Let $G(V, E)$ be any weighted connected graph.

   (a) If C is any cycle of G, then show that the heaviest edge of C cannot belong to a minimum-cost spanning tree of G.

   (b) Assume that F is a forest that is a subgraph of G. Show that any F-heavy edge of G cannot belong to a minimum-cost spanning tree of G.

8. By considering the complete graph with n vertices, show that the number of spanning trees in an n vertex graph can be greater than $2^{n-1}-2$.

4.6 OPTIMAL STORAGE ON TAPES

There are n programs that are to be stored on a computer tape of length l. Associated with each program i is a length $l_i, 1 \leq i \leq n$. Clearly, all programs can be stored on the tape if and only if the sum of the lengths of

the programs is at most l. We assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence, if the programs are stored in the order $I = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \le k \le j} l_{i_k}$. If all programs are retrieved equally often, then the expected or *mean retrieval time* (MRT) is $(1/n) \sum_{1 \le j \le n} t_j$. In the optimal storage on tape problem, we are required to find a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized. This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing $d(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{i_k}$.

Example 4.8 Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings. These orderings and their respective d values are:

ordering I	$d(I)$		
$1, 2, 3$	$5 + 5 + 10 + 5 + 10 + 3$	$=$	38
$1, 3, 2$	$5 + 5 + 3 + 5 + 3 + 10$	$=$	31
$2, 1, 3$	$10 + 10 + 5 + 10 + 5 + 3$	$=$	43
$2, 3, 1$	$10 + 10 + 3 + 10 + 3 + 5$	$=$	41
$3, 1, 2$	$3 + 3 + 5 + 3 + 5 + 10$	$=$	29
$3, 2, 1$	$3 + 3 + 10 + 3 + 10 + 5$	$=$	34

The optimal ordering is $3, 1, 2$. □

A greedy approach to building the required permutation would choose the next program on the basis of some optimization measure. One possible measure would be the d value of the permutation constructed so far. The next program to be stored on the tape would be one that minimizes the increase in d. If we have already constructed the permutation $i_1, i_2, \ldots, i_r$, then appending program j gives the permutation $i_1, i_2, \ldots, i_r, i_{r+1} = j$. This increases the d value by $\sum_{1 \le k \le r} l_{i_k} + l_j$. Since $\sum_{1 \le k \le r} l_{i_k}$ is fixed and independent of j, we trivially observe that the increase in d is minimized if the next program chosen is the one with the least length from among the remaining programs.

The greedy algorithm resulting from the above discussion is so simple that we won't bother to write it out. The greedy method simply requires us to store the programs in nondecreasing order of their lengths. This ordering can be carried out in $O(n \log n)$ time using an efficient sorting algorithm (e.g., heap sort from Chapter 2). For the programs of Example 4.8, note that the permutation that yields an optimal solution is the one in which the programs are in nondecreasing order of their lengths. Theorem 4.8 shows that the MRT is minimized when programs are stored in this order.

Theorem 4.8 If $l_1 \leq l_2 \leq \cdots \leq l_n$, then the ordering $i_j = j, 1 \leq j \leq n$, minimizes

$$\sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j}$$

over all possible permutations of the i_j.

Proof: Let $I = i_1, i_2, \ldots, i_n$ be any permutation of the index set $\{1, 2, \ldots, n\}$. Then

$$d(I) = \sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j} = \sum_{k=1}^{n} (n - k + 1) l_{i_k}$$

If there exist a and b such that $a < b$ and $l_{i_a} > l_{i_b}$, then interchanging i_a and i_b results in a permutation I' with

$$d(I') = \left[\sum_{\substack{k \\ k \neq a \\ k \neq b}} (n - k + 1) l_{i_k} \right] + (n - a + 1) l_{i_b} + (n - b + 1) l_{i_a}$$

Subtracting $d(I')$ from $d(I)$, we obtain

$$\begin{aligned} d(I) - d(I') &= (n - a + 1)(l_{i_a} - l_{i_b}) + (n - b + 1)(l_{i_b} - l_{i_a}) \\ &= (b - a)(l_{i_a} - l_{i_b}) \\ &> 0 \end{aligned}$$

Hence, no permutation that is not in nondecreasing order of the l_i's can have minimum d. It is easy to see that all permutations in nondecreasing order of the l_i's have the same d value. Hence, the ordering defined by $i_j = j, 1 \leq j \leq n$, minimizes the d value. □

The tape storage problem can be extended to several tapes. If there are $m > 1$ tapes, $T_0, \ldots, T_{m-1}$, then the programs are to be distributed over these tapes. For each tape a storage permutation is to be provided. If I_j is the storage permutation for the subset of programs on tape j, then $d(I_j)$ is as defined earlier. The *total retrieval time* (TD) is $\sum_{0 \leq j \leq m-1} d(I_j)$. The objective is to store the programs in such a way as to minimize TD.

The obvious generalization of the solution for the one-tape case is to consider the programs in nondecreasing order of l_i's. The program currently

```
1   Algorithm Store(n, m)
2   // n is the number of programs and m the number of tapes.
3   {
4       j := 0; // Next tape to store on
5       for i := 1 to n do
6       {
7           write ("append program", i,
8               "to permutation for tape", j);
9           j := (j + 1) mod m;
10      }
11  }
```

Algorithm 4.12 Assigning programs to tapes

being considered is placed on the tape that results in the minimum increase in TD. This tape will be the one with the least amount of tape used so far. If there is more than one tape with this property, then the one with the smallest index can be used. If the jobs are initially ordered so that $l_1 \leq l_2 \leq \cdots \leq l_n$, then the first m programs are assigned to tapes $T_0, \ldots, T_{m-1}$ respectively. The next m programs will be assigned to tapes $T_0, \ldots, T_{m-1}$ respectively. The general rule is that program i is stored on tape $T_{i \bmod m}$. On any given tape the programs are stored in nondecreasing order of their lengths. Algorithm 4.12 presents this rule in pseudocode. It assumes that the programs are ordered as above. It has a computing time of $\Theta(n)$ and does not need to know the program lengths. Theorem 4.9 proves that the resulting storage pattern is optimal.

Theorem 4.9 If $l_1 \leq l_2 \leq \cdots \leq l_n$, then Algorithm 4.12 generates an optimal storage pattern for m tapes.

Proof: In any storage pattern for m tapes, let r_i be one greater than the number of programs following program i on its tape. Then the total retrieval time TD is given by

$$TD = \sum_{i=1}^{n} r_i l_i$$

In any given storage pattern, for any given n, there can be at most m programs for which $r_i = j$. From Theorem 4.8 it follows that TD is minimized if the m longest programs have $r_i = 1$, the next m longest programs have

$r_i = 2$, and so on. When programs are ordered by length, that is, $l_1 \leq l_2 \leq \cdots \leq l_n$, then this minimization criteria is satisfied if $r_i = \lceil (n-i+1)/m \rceil$. Observe that Algorithm 4.12 results in a storage pattern with these r_i's. □

The proof of Theorem 4.9 shows that there are many storage patterns that minimize TD. If we compute $r_i = \lceil (n-i+1)/m \rceil$ for each program i, then so long as all programs with the same r_i are stored on different tapes and have $r_i - 1$ programs following them, TD is the same. If n is a multiple of m, then there are at least $(m!)^{n/m}$ storage patterns that minimize TD. Algorithm 4.12 produces one of these.

EXERCISES

1. Find an optimal placement for 13 programs on three tapes T_0, T_1, and T_2, where the programs are of lengths $12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10$, and 6.

2. Show that replacing the code of Algorithm 4.12 by

```
for i := 1 to n do
    write ("append program", i, "to permutation for
        tape", (i - 1) mod m);
```

   does not affect the output.

3. Let $P_1, P_2, \ldots, P_n$ be a set of n programs that are to be stored on a tape of length l. Program P_i requires a_i amount of tape. If $\sum a_i \leq l$, then clearly all the programs can be stored on the tape. So, assume $\sum a_i > l$. The problem is to select a maximum subset Q of the programs for storage on the tape. (A maximum subset is one with the maximum number of programs in it). A greedy algorithm for this problem would build the subset Q by including programs in nondecreasing order of a_i.

   (a) Assume the P_i are ordered such that $a_1 \leq a_2 \leq \cdots \leq a_n$. Write a function for the above strategy. Your function should output an array $s[1:n]$ such that $s[i] = 1$ if P_i is in Q and $s[i] = 0$ otherwise.

   (b) Show that this strategy always finds a maximum subset Q such that $\sum_{P_i \in Q} a_i \leq l$.

   (c) Let Q be the subset obtained using the above greedy strategy. How small can the tape utilization ratio $(\sum_{P_i \in Q} a_i)/l$ get?

   (d) Suppose the objective now is to determine a subset of programs that maximizes the tape utilization ratio. A greedy approach

would be to consider programs in nonincreasing order of a_i. If there is enough space left on the tape for P_i, then it is included in Q. Assume the programs are ordered so that $a_1 \geq a_2 \geq \cdots \geq a_n$. Write a function incorporating this strategy. What is its time and space complexity?

(e) Show that the strategy of part (d) doesn't necessarily yield a subset that maximizes $(\sum_{P_i \in Q} a_i)/l$. How small can this ratio get? Prove your bound.

4. Assume n programs of lengths $l_1, l_2, \ldots, l_n$ are to be stored on a tape. Program i is to be retrieved with frequency f_i. If the programs are stored in the order $i_1, i_2, \ldots, i_n$, the *expected retrieval time* (ERT) is

$$\left[\sum_j (f_{i_j} \sum_{k=1}^{j} l_{i_k})\right] / \sum f_i$$

(a) Show that storing the programs in nondecreasing order of l_i does not necessarily minimize the ERT.

(b) Show that storing the programs in nonincreasing order of f_i does not necessarily minimize the ERT.

(c) Show that the ERT is minimized when the programs are stored in nonincreasing order of f_i/l_i.

5. Consider the tape storage problem of this section. Assume that two tapes $T1$ and $T2$, are available and we wish to distribute n given programs of lengths $l_1, l_2, \ldots, l_n$ onto these two tapes in such a manner that the maximum retrieval time is minimized. That is, if A and B are the sets of programs on the tapes $T1$ and $T2$ respectively, then we wish to choose A and B such that $\max \{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$ is minimized. A possible greedy approach to obtaining A and B would be to start with A and B initially empty. Then consider the programs one at a time. The program currently being considered is assigned to set A if $\sum_{i \in A} l_i = \min \{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$; otherwise it is assigned to B. Show that this does not guarantee optimal solutions even if $l_1 \leq l_2 \leq \cdots \leq l_n$. Show that the same is true if we require $l_1 \geq l_2 \geq \cdots \geq l_n$.

4.7 OPTIMAL MERGE PATTERNS

In Section 3.4 we saw that two sorted files containing n and m records respectively could be merged together to obtain one sorted file in time $O(n+m)$. When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs. Thus, if

files x_1, x_2, x_3, and x_4 are to be merged, we could first merge x_1 and x_2 to get a file y_1. Then we could merge y_1 and x_3 to get y_2. Finally, we could merge y_2 and x_4 to get the desired sorted file. Alternatively, we could first merge x_1 and x_2 getting y_1, then merge x_3 and x_4 and get y_2, and finally merge y_1 and y_2 and get the desired sorted file. Given n sorted files, there are many ways in which to pairwise merge them into a single sorted file. Different pairings require differing amounts of computing time. The problem we address ourselves to now is that of determining an optimal way (one requiring the fewest comparisons) to pairwise merge n sorted files. Since this problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.

Example 4.9 The files x_1, x_2, and x_3 are three sorted files of length 30, 20, and 10 records each. Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If, instead, we first merge x_2 and x_3 (taking 30 moves) and then x_1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first. □

A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n-record file and an m-record file requires possibly $n + m$ record moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together. Thus, if we have five files $(x_1, \ldots, x_5)$ with sizes $(20, 30, 10, 5, 30)$, our greedy rule would generate the following merge pattern: merge x_4 and x_3 to get z_1 ($|z_1| = 15$), merge z_1 and x_1 to get z_2 ($|z_2| = 35$), merge x_2 and x_5 to get z_3 ($|z_3| = 60$), and merge z_2 and z_3 to get the answer z_4. The total number of record moves is 205. One can verify that this is an optimal merge pattern for the given problem instance.

The merge pattern such as the one just described will be referred to as a *two-way merge pattern* (each merge step involves the merging of two files). The two-way merge patterns can be represented by binary merge trees. Figure 4.11 shows a binary merge tree representing the optimal merge pattern obtained for the above five files. The leaf nodes are drawn as squares and represent the given five files. These nodes are called *external nodes*. The remaining nodes are drawn as circles and are called *internal nodes*. Each internal node has exactly two children, and it represents the file obtained by merging the files represented by its two children. The number in each node is the length (i.e., the number of records) of the file represented by that node.

The external node x_4 is at a distance of 3 from the root node z_4 (a node at level i is at a distance of $i - 1$ from the root). Hence, the records of file x_4 are moved three times, once to get z_1, once again to get z_2, and finally one more time to get z_4. If d_i is the distance from the root to the external

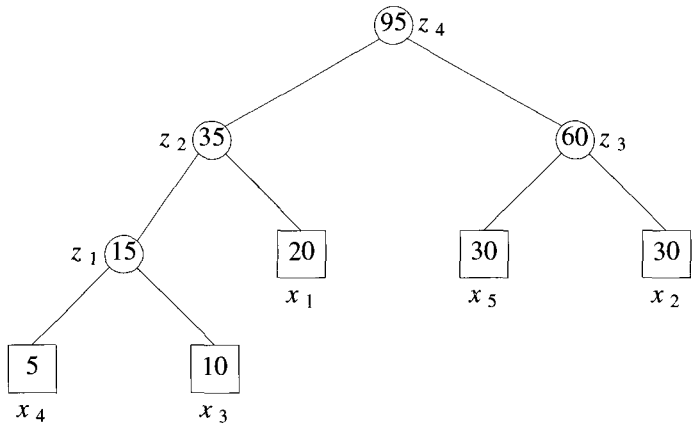


Figure 4.11 Binary merge tree representing a merge pattern

node for file x_i and q_i, the length of x_i is then the total number of record moves for this binary merge tree is

$$\sum_{i=1}^{n} d_i q_i$$

This sum is called the *weighted external path length* of the tree.

An optimal two-way merge pattern corresponds to a binary merge tree with minimum weighted external path length. The function Tree of Algorithm 4.13 uses the greedy rule stated earlier to obtain a two-way merge tree for n files. The algorithm has as input a list *list* of n trees. Each node in a tree has three fields, *lchild*, *rchild*, and *weight*. Initially, each tree in *list* has exactly one node. This node is an external node and has *lchild* and *rchild* fields zero whereas *weight* is the length of one of the n files to be merged. During the course of the algorithm, for any tree in *list* with root node t, $t \rightarrow weight$ is the length of the merged file it represents ($t \rightarrow weight$ equals the sum of the lengths of the external nodes in tree t). Function Tree uses two functions, Least(*list*) and Insert(*list*, t). Least(*list*) finds a tree in *list* whose root has least *weight* and returns a pointer to this tree. This tree is removed from *list*. Insert(*list*, t) inserts the tree with root t into *list*. Theorem 4.10 shows that Tree (Algorithm 4.13) generates an optimal two-way merge tree.

```
    treenode = record {
        treenode * lchild; treenode * rchild;
        integer weight;
    };

1   Algorithm Tree(n)
2   // list is a global list of n single node
3   // binary trees as described above.
4   {
5       for i := 1 to n - 1 do
6       {
7           pt := new treenode; // Get a new tree node.
8           (pt → lchild) := Least(list); // Merge two trees with
9           (pt → rchild) := Least(list); // smallest lengths.
10          (pt → weight) := ((pt → lchild) → weight)
11                  +((pt → rchild) → weight);
12          Insert(list, pt);
13      }
14      return Least(list); // Tree left in list is the merge tree.
15  }
```

Algorithm 4.13 Algorithm to generate a two-way merge tree

Example 4.10 Let us see how algorithm Tree works when *list* initially represents six files with lengths $(2, 3, 5, 7, 9, 13)$. Figure 4.12 shows *list* at the end of each iteration of the **for** loop. The binary merge tree that results at the end of the algorithm can be used to determine which files are merged. Merging is performed on those files which are lowest (have the greatest depth) in the tree. □

The main **for** loop in Algorithm 4.13 is executed $n - 1$ times. If *list* is kept in nondecreasing order according to the *weight* value in the roots, then Least(*list*) requires only $O(1)$ time and Insert($list, t$) can be done in $O(n)$ time. Hence the total time taken is $O(n^2)$. In case *list* is represented as a minheap in which the root value is less than or equal to the values of its children (Section 2.4), then Least(*list*) and Insert($list, t$) can be done in $O(\log n)$ time. In this case the computing time for Tree is $O(n \log n)$. Some speedup may be obtained by combining the Insert of line 12 with the Least of line 9.

Theorem 4.10 If *list* initially contains $n \geq 1$ single node trees with *weight* values $(q_1, q_2, \ldots, q_n)$, then algorithm Tree generates an optimal two-way merge tree for n files with these lengths.

Proof: The proof is by induction on n. For $n = 1$, a tree with no internal nodes is returned and this tree is clearly optimal. For the induction hypothesis, assume the algorithm generates an optimal two-way merge tree for all $(q_1, q_2, \ldots, q_m)$, $1 \leq m < n$. We show that the algorithm also generates optimal trees for all $(q_1, q_2, \ldots, q_n)$. Without loss of generality, we can assume that $q_1 \leq q_2 \leq \cdots \leq q_n$ and q_1 and q_2 are the values of the *weight* fields of the trees found by algorithm Least in lines 8 and 9 during the first iteration of the **for** loop. Now, the subtree T of Figure 4.13 is created. Let T' be an optimal two-way merge tree for $(q_1, q_2, \ldots, q_n)$. Let p be an internal node of maximum distance from the root. If the children of p are not q_1 and q_2, then we can interchange the present children with q_1 and q_2 without increasing the weighted external path length of T'. Hence, T is also a subtree in an optimal merge tree. If we replace T in T' by an external node with weight $q_1 + q_2$, then the resulting tree T'' is an optimal merge tree for $(q_1 + q_2, q_3, \ldots, q_n)$. From the induction hypothesis, after replacing T by the external node with value $q_1 + q_2$, function Tree proceeds to find an optimal merge tree for $(q_1 + q_2, q_3, \ldots, q_n)$. Hence, Tree generates an optimal merge tree for $(q_1, q_2, \ldots, q_n)$. □

The greedy method to generate merge trees also works for the case of k-ary merging. In this case the corresponding merge tree is a k-ary tree. Since all internal nodes must have degree k, for certain values of n there is no corresponding k-ary merge tree. For example, when $k = 3$, there is no k-ary merge tree with $n = 2$ external nodes. Hence, it is necessary to introduce a certain number of dummy external nodes. Each dummy node is assigned a q_i of zero. This dummy value does not affect the weighted external path length of the resulting k-ary tree. Exercise 2 shows that a k-ary tree with all internal nodes having degree k exists only when the number of external nodes n satisfies the equality $n \bmod (k-1) = 1$. Hence, at most $k-2$ dummy nodes have to be added. The greedy rule to generate optimal merge trees is: at each step choose k subtrees with least length for merging. Exercise 3 proves the optimality of this rule.

Huffman Codes

Another application of binary trees with minimal weighted external path length is to obtain an optimal set of codes for messages $M_1, \ldots, M_{n+1}$. Each code is a binary string that is used for transmission of the corresponding message. At the receiving end the code is decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages.

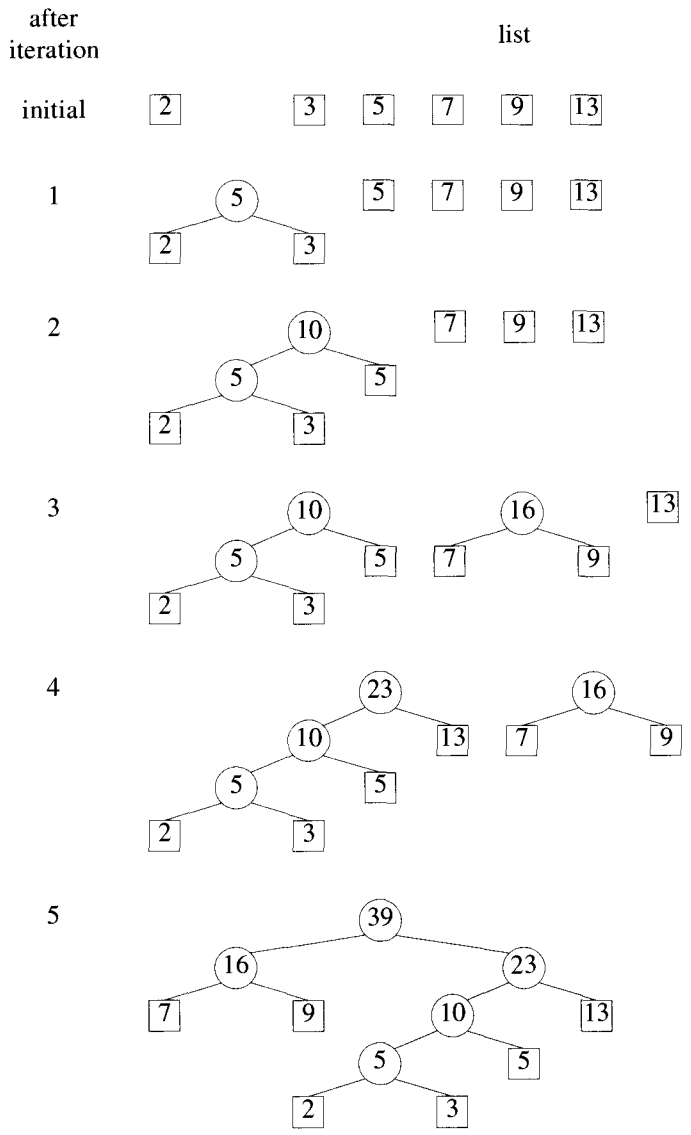


Figure 4.12 Trees in *list* of Tree for Example 4.10

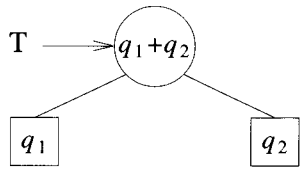


Figure 4.13 The simplest binary merge tree

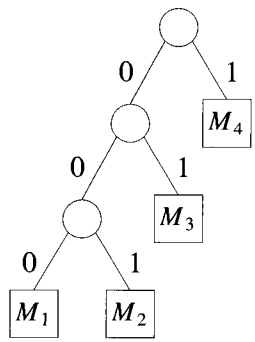


Figure 4.14 Huffman codes

The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of Figure 4.14 corresponds to codes 000, 001, 01, and 1 for messages M_1, M_2, M_3, and M_4 respectively. These codes are called Huffman codes. The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is $\sum_{1 \le i \le n+1} q_i d_i$, where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length! Note that $\sum_{1 \le i \le n+1} q_i d_i$ is also the expected length of a transmitted message. Hence the code that minimizes expected decode time also minimizes the expected length of a message.

EXERCISES

1. Find an optimal binary merge pattern for ten files whose lengths are $28, 32, 12, 5, 84, 53, 91, 35, 3,$ and 11.

2. (a) Show that if all internal nodes in a tree have degree k, then the number n of external nodes is such that $n \bmod (k-1) = 1$.
   (b) Show that for every n such that $n \bmod (k-1) = 1$, there exists a k-ary tree T with n external nodes (in a k-ary tree all nodes have degree at most k). Also show that all internal nodes of T have degree k.

3. (a) Show that if $n \bmod (k-1) = 1$, then the greedy rule described following Theorem 4.10 generates an optimal k-ary merge tree for all $(q_1, q_2, \ldots, q_n)$.
   (b) Draw the optimal three-way merge tree obtained using this rule when $(q_1, q_2, \ldots, q_{11}) = (3, 7, 8, 9, 15, 16, 18, 20, 23, 25, 28)$.

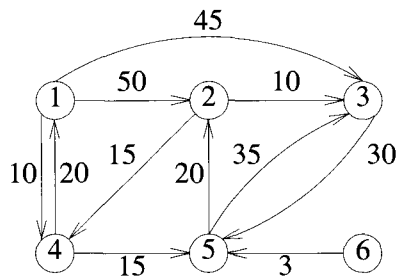
4. Obtain a set of optimal Huffman codes for the messages $(M_1, \ldots, M_7)$ with relative frequencies $(q_1, \ldots, q_7) = (4, 5, 7, 8, 10, 12, 20)$. Draw the decode tree for this set of codes.

5. Let T be a decode tree. An optimal decode tree minimizes $\sum q_i d_i$. For a given set of q's, let D denote all the optimal decode trees. For any tree $T \in D$, let $L(T) = \max\{d_i\}$ and let $\mathrm{SL}(T) = \sum d_i$. Schwartz has shown that there exists a tree $T^* \in D$ such that $L(T^*) = \min_{T \in D}\{L(T)\}$ and $\mathrm{SL}(T^*) = \min_{T \in D}\{SL(T)\}$.

   (a) For $(q_1, \ldots, q_8) = (1, 1, 2, 2, 4, 4, 4, 4)$ obtain trees $T1$ and $T2$ such that $L(T1) > L(T2)$.
   (b) Using the data of a, obtain $T1$ and $T2 \in D$ such that $L(T1) = L(T2)$ but $SL(T1) > SL(T2)$.
   (c) Show that if the subalgorithm Least used in algorithm Tree is such that in case of a tie it returns the tree with least depth, then Tree generates a tree with the properties of T^*.

4.8 SINGLE-SOURCE SHORTEST PATHS

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:



(a) Graph

Path	*Length*
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

Figure 4.15 Graph and shortest paths from vertex 1 to all destinations

- Is there a path from A to B?
- If there is more than one path from A to B, which is the shortest path?

The problems defined by these questions are special cases of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the *source*, and the last vertex the *destination*. The graphs are digraphs to allow for one-way streets. In the problem we consider, we are given a directed graph $G = (V, E)$, a weighting function *cost* for the edges of G, and a source vertex v_0. The problem is to determine the shortest paths from v_0 to *all* the remaining vertices of G. It is assumed that all the weights are positive. The shortest path between v_0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example 4.11 Consider the directed graph of Figure 4.15(a). The numbers on the edges are the weights. If node 1 is the source vertex, then the shortest path from 1 to 2 is $1, 4, 5, 2$. The length of this path is $10 + 15 + 20 = 45$. Even though there are three edges on this path, it is shorter than the path $1, 2$ which is of length 50. There is no path from 1 to 6. Figure 4.15(b) lists the shortest paths from node 1 to nodes $4, 5, 2$, and 3, respectively. The paths have been listed in nondecreasing order of path length. □

To formulate a greedy-based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by

one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way (and also a systematic way) to generate the shortest paths from v_0 to the remaining vertices is to generate these paths in nondecreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on. For the graph of Figure 4.15(a) the nearest vertex to $v_0 = 1$ is 4 ($cost[1,4] = 10$). The path $1,4$ is the first path generated. The second nearest vertex to node 1 is 5 and the distance between 1 and 5 is 25. The path $1,4,5$ is the next path generated. In order to generate the shortest paths in this order, we need to be able to determine (1) the next vertex to which a shortest path must be generated and (2) a shortest path to this vertex. Let S denote the set of vertices (including v_0) to which the shortest paths have already been generated. For w not in S, let $dist[w]$ be the length of the shortest path starting from v_0, going through only those vertices that are in S, and ending at w. We observe that:

1. If the next shortest path is to vertex u, then the path begins at v_0, ends at u, and goes through only those vertices that are in S. To prove this, we must show that all the intermediate vertices on the shortest path to u are in S. Assume there is a vertex w on this path that is not in S. Then, the v_0 to u path also contains a path from v_0 to w that is of length less than the v_0 to u path. By assumption the shortest paths are being generated in nondecreasing order of path length, and so the shorter path v_0 to w must already have been generated. Hence, there can be no intermediate vertex that is not in S.

2. The destination of the next path generated must be that of vertex u which has the minimum distance, $dist[u]$, among all vertices not in S. This follows from the definition of $dist$ and observation 1. In case there are several vertices not in S with the same $dist$, then any of these may be selected.

3. Having selected a vertex u as in observation 2 and generated the shortest v_0 to u path, vertex u becomes a member of S. At this point the length of the shortest paths starting at v_0, going though vertices only in S, and ending at a vertex w not in S may decrease; that is, the value of $dist[w]$ may change. If it does change, then it must be due to a shorter path starting at v_0 and going to u and then to w. The intermediate vertices on the v_0 to u path and the u to w path must all be in S. Further, the v_0 to u path must be the shortest such path; otherwise $dist[w]$ is not defined properly. Also, the u to w path can be chosen so as not to contain any intermediate vertices. Therefore,

we can conclude that if $dist[w]$ is to change (i.e., decrease), then it is because of a path from v_0 to u to w, where the path from v_0 to u is the shortest such path and the path from u to w is the edge $\langle u, w \rangle$. The length of this path is $dist[u] + cost[u, w]$.

The above observations lead to a simple Algorithm 4.14 for the single-source shortest path problem. This algorithm (known as Dijkstra's algorithm) only determines the lengths of the shortest paths from v_0 to all other vertices in G. The generation of the paths requires a minor extension to this algorithm and is left as an exercise. In the function ShortestPaths (Algorithm 4.14) it is assumed that the n vertices of G are numbered 1 through n. The set S is maintained as a bit array with $S[i] = 0$ if vertex i is not in S and $S[i] = 1$ if it is. It is assumed that the graph itself is represented by its cost adjacency matrix with $cost[i, j]$'s being the weight of the edge $\langle i, j \rangle$. The weight $cost[i, j]$ is set to some large number, ∞, in case the edge $\langle i, j \rangle$ is not in $E(G)$. For $i = j$, $cost[i, j]$ can be set to any nonnegative number without affecting the outcome of the algorithm.

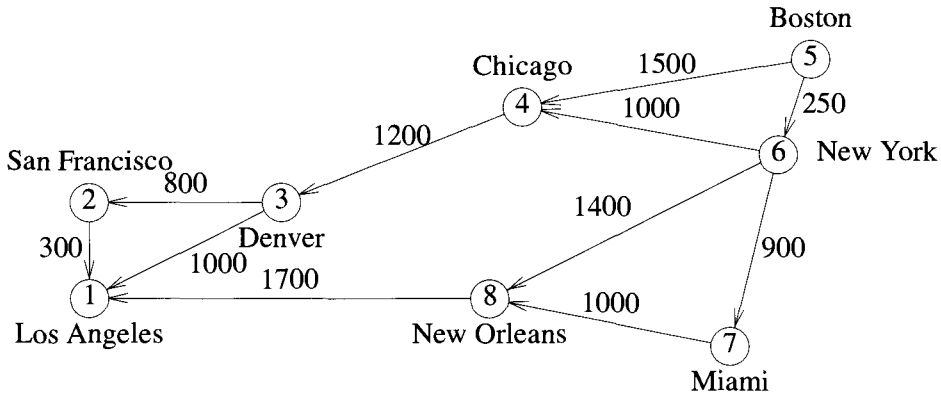
From our earlier discussion, it is easy to see that the algorithm is correct. The time taken by the algorithm on a graph with n vertices is $O(n^2)$. To see this, note that the **for** loop of line 7 in Algorithm 4.14 takes $\Theta(n)$ time. The **for** loop of line 12 is executed $n - 2$ times. Each execution of this loop requires $O(n)$ time at lines 15 and 16 to select the next vertex and again at the **for** loop of line 18 to update $dist$. So the total time for this loop is $O(n^2)$. In case a list t of vertices currently not in s is maintained, then the number of nodes on this list would at any time be $n - num$. This would speed up lines 15 and 16 and the **for** loop of line 18, but the asymptotic time would remain $O(n^2)$. This and other variations of the algorithm are explored in the exercises.

Any shortest path algorithm must examine each edge in the graph at least once since any of the edges could be in a shortest path. Hence, the minimum possible time for such an algorithm would be $\Omega(|E|)$. Since cost adjacency matrices were used to represent the graph, it takes $O(n^2)$ time just to determine which edges are in G, and so any shortest path algorithm using this representation must take $\Omega(n^2)$ time. For this representation then, algorithm ShortestPaths is optimal to within a constant factor. If a change to adjacency lists is made, the overall frequency of the **for** loop of line 18 can be brought down to $O(|E|)$ (since $dist$ can change only for vertices adjacent from u). If $V - S$ is maintained as a red-black tree (see Section 2.4.2), each execution of lines 15 and 16 takes $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). Each update in line 21 takes $O(\log n)$ time as well (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O((n + |E|) \log n)$.

```
1   Algorithm ShortestPaths(v, cost, dist, n)
2   // dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
3   // path from vertex v to vertex j in a digraph G with n
4   // vertices. dist[v] is set to zero. G is represented by its
5   // cost adjacency matrix cost[1 : n, 1 : n].
6   {
7       for i := 1 to n do
8       { // Initialize S.
9           S[i] := false; dist[i] := cost[v, i];
10      }
11      S[v] := true; dist[v] := 0.0; // Put v in S.
12      for num := 2 to n − 1 do
13      {
14          // Determine n − 1 paths from v.
15          Choose u from among those vertices not
16          in S such that dist[u] is minimum;
17          S[u] := true; // Put u in S.
18          for (each w adjacent to u with S[w] = false) do
19              // Update distances.
20              if (dist[w] > dist[u] + cost[u, w])) then
21                      dist[w] := dist[u] + cost[u, w];
22      }
23  }
```

Algorithm 4.14 Greedy algorithm to generate shortest paths

Example 4.12 Consider the eight vertex digraph of Figure 4.16(a) with cost adjacency matrix as in Figure 4.16(b). The values of *dist* and the vertices selected at each iteration of the **for** loop of line 12 in Algorithm 4.14 for finding all the shortest paths from Boston are shown in Figure 4.17. To begin with, S contains only Boston. In the first iteration of the **for** loop (that is, for $num = 2$), the city u that is not in S and whose $dist[u]$ is minimum is identified to be New York. New York enters the set S. Also the $dist[\,]$ values of Chicago, Miami, and New Orleans get altered since there are shorter paths to these cities via New York. In the next iteration of the **for** loop, the city that enters S is Miami since it has the smallest $dist[\,]$ value from among all the nodes not in S. None of the $dist[\,]$ values are altered. The algorithm continues in a similar fashion and terminates when only seven of the eight vertices are in S. By the definition of *dist*, the distance of the last vertex, in this case Los Angeles, is correct as the shortest path from Boston to Los Angeles can go through only the remaining six vertices. □



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Figure 4.16 Figures for Example 4.12

One can easily verify that the edges on the shortest paths from a vertex v to all remaining vertices in a connected undirected graph G form a spanning tree of G. This spanning tree is called a *shortest-path spanning tree*. Clearly, this spanning tree may be different for different root vertices v. Figure 4.18 shows a graph G, its minimum-cost spanning tree, and a shortest-path spanning tree from vertex 1.

Iteration	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

Figure 4.17 Action of ShortestPaths

EXERCISES

1. Use algorithm ShortestPaths to obtain in nondecreasing order the lengths of the shortest paths from vertex 1 to all remaining vertices in the digraph of Figure 4.19.

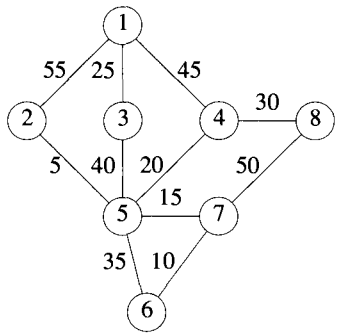
2. Using the directed graph of Figure 4.20 explain why ShortestPaths will not work properly. What is the shortest path between vertices v_1 and v_7 ?

3. Rewrite algorithm ShortestPaths under the following assumptions:

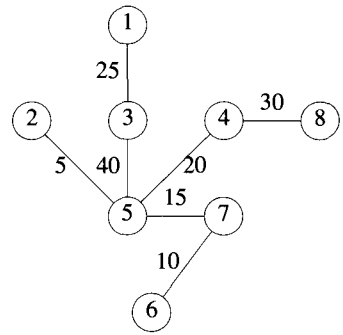
   (a) G is represented by its adjacency lists. The head nodes are HEAD(1),..., HEAD(n) and each list node has three fields: VERTEX, COST, and LINK. COST is the length of the corresponding edge and n the number of vertices in G.

   (b) Instead of representing S, the set of vertices to which the shortest paths have already been found, the set $T = V(G) - S$ is represented using a linked list. What can you say about the computing time of your new algorithm relative to that of ShortestPaths?

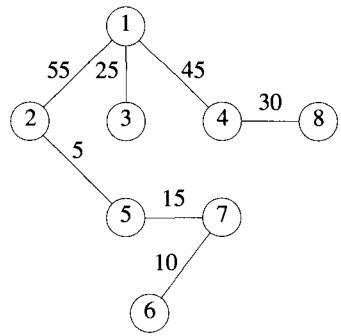
4. Modify algorithm ShortestPaths so that it obtains the shortest paths in addition to the lengths of these paths. What is the computing time of your algorithm?



(a) A Graph



(b) Minimum cost spanning tree



(c) Shortest path spanning tree from vertex 1.

Figure 4.18 Graphs and spanning trees

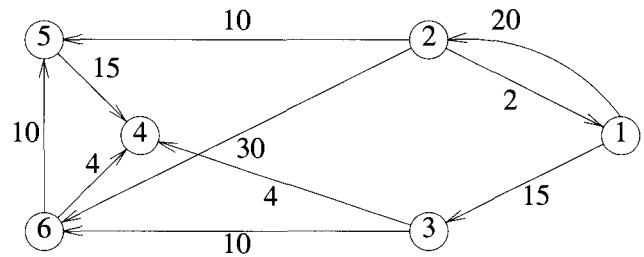


Figure 4.19 Directed graph

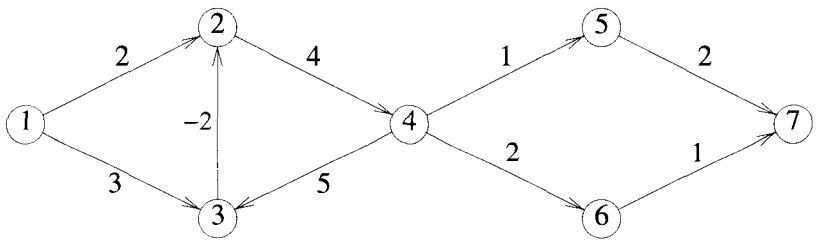


Figure 4.20 Another directed graph

4.9 REFERENCES AND READINGS

The linear time algorithm in Section 4.3 for the tree vertex splitting problem can be found in "Vertex upgrading problems for VLSI," by D. Paik, Ph.D. thesis, Department of Computer Science, University of Minnesota, October 1991.

The two greedy methods for obtaining minimum-cost spanning trees are due to R. C. Prim and J. B. Kruskal, respectively.

An $O(e \log \log v)$ time spanning tree algorithm has been given by A. C. Yao.

The optimal randomized algorithm for minimum-cost spanning trees presented in this chapter appears in "A randomized linear-time algorithm for finding minimum spanning trees," by P. N. Klein and R. E. Tarjan, in *Proceedings of the 26th Annual Symposium on Theory of Computing*, 1994, pp. 9–15. See also "A randomized linear-time algorithm to find minimum spanning trees," by D. R. Karger, P. N. Klein, and R. E. Tarjan, *Journal of the ACM* 42, no. 2 (1995): 321–328.

Proof of Lemma 4.3 can be found in "Verification and sensitivity analysis of minimum spanning trees in linear time," by B. Dixon, M. Rauch, and R. E. Tarjan, *SIAM Journal on Computing* 21 (1992): 1184–1192, and in "A simple minimum spanning tree verification algorithm," by V. King, *Proceedings of the Workshop on Algorithms and Data Structures*, 1995.

A very nearly linear time algorithm for minimum-cost spanning trees appears in "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," by H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, *Combinatorica* 6 (1986): 109–122.

A linear time algorithm for minimum-cost spanning trees on a stronger model where the edge weights can be manipulated in their binary form is given in "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," by M. Fredman and D. E. Willard, in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, 1990, pp. 719–725.

The greedy method developed here to optimally store programs on tapes was first devised for a machine scheduling problem. In this problem n jobs have to be scheduled on m processors. Job i takes t_i amount of time. The time at which a job finishes is the sum of the job times for all jobs preceding and including job i. The average finish time corresponds to the mean access time for programs on tapes. The $(m!)^{n/m}$ schedules referred to in Theorem 4.9 are known as SPT (shortest processing time) schedules. The rule to generate SPT schedules as well as the rule of Exercise 4 (Section 4.6) are due to W. E. Smith.

The greedy algorithm for generating optimal merge trees is due to D. Huffman.

For a given set $\{q_1, \ldots, q_n\}$ there are many sets of Huffman codes minimizing $\sum q_i d_i$. From amongst these code sets there is one that has minimum $\sum d_i$ and minimum max $\{d_i\}$. An algorithm to obtain this code set was given by E. S. Schwartz.

The shortest-path algorithm of the text is due to E. W. Dijkstra.

For planar graphs, the shortest-path problem can be solved in linear time as has been shown in "Faster shortest-path algorithms for planar graphs," by P. Klein, S. Rao, and M. Rauch, in *Proceedings of the ACM Symposium on Theory of Computing*, 1994.

The relationship between greedy methods and matroids is discussed in *Combinatorial Optimization*, by E. Lawler, Holt, Rinehart and Winston, 1976.

4.10 ADDITIONAL EXERCISES

1. [Coin changing] Let $A_n = \{a_1, a_2, \ldots, a_n\}$ be a finite set of distinct coin types (for example, $a_1 = 50¢$, $a_2 = 25¢$, $a_3 = 10¢$, and so on.) We can assume each a_i is an integer and $a_1 > a_2 > \cdots > a_n$. Each type is available in unlimited quantity. The coin-changing problem is to make up an exact amount C using a minimum total number of coins. C is an integer > 0.

(a) Show that if $a_n \neq 1$, then there exists a finite set of coin types and a C for which there is no solution to the coin-changing problem.

(b) Show that there is always a solution when $a_n = 1$.

(c) When $a_n = 1$, a greedy solution to the problem makes change by using the coin types in the order $a_1, a_2, \ldots, a_n$. When coin type a_i is being considered, as many coins of this type as possible are given. Write an algorithm based on this strategy. Show that this algorithm doesn't necessarily generate solutions that use the minimum total number of coins.

(d) Show that if $A_n = \{k^{n-1}, k^{n-2}, \ldots, k^0\}$ for some $k > 1$, then the greedy method of part (c) always yields solutions with a minimum number of coins.

2. [Set cover] You are given a family S of m sets $S_i, 1 \leq i \leq m$. Denote by $|A|$ the size of set A. Let $|S_i| = j_i$; that is, $S_i = \{s_1, s_2, \ldots, s_{j_i}\}$. A subset $T = \{T_1, T_2, \ldots, T_k\}$ of S is a family of sets such that for each $i, 1 \leq i \leq k$, $T_i = S_r$ for some $r, 1 \leq r \leq m$. The subset T is a *cover* of S iff $\cup T_i = \cup S_i$. The size of T, $|T|$, is the number of sets in T. A minimum cover of S is a cover of smallest size. Consider the following greedy strategy: build T iteratively, at the kth iteration $T = \{T_1, \ldots, T_{k-1}\}$, now add to T a set S_j from S that contains the largest number of elements not already in T, and stop when $\cup T_i = \cup S_i$.

(a) Assume that $\cup S_i = \{1, 2, \ldots, n\}$ and $m < n$. Using the strategy outlined above, write an algorithm to obtain set covers. How much time and space does your algorithm require?

(b) Show that the greedy strategy above doesn't necessarily obtain a minimum set cover.

(c) Suppose now that a minimum cover is defined to be one for which $\sum_{i=1}^{k} |T_i|$ is minimum. Does the above strategy always find a minimum cover?

3. [Node cover] Let $G = (V, E)$ be an undirected graph. A node cover of G is a subset U of the vertex set V such that every edge in E is incident to at least one vertex in U. A minimum node cover is one with the fewest number of vertices. Consider the following greedy algorithm for this problem:

```
1   Algorithm Cover(V, E)
2   {
3       U := ∅;
4       repeat
5       {
6           Let q be a vertex from V of maximum degree;
7           Add q to U; Eliminate q from V;
8           E := E − {(x, y) such that x = q or y = q};
9       } until (E = ∅); // U is the node cover.
10  }
```

Does this algorithm always generate a minimum node cover?

4. [Traveling salesperson] Let G be a directed graph with n vertices. Let $length(u, v)$ be the length of the edge $\langle u, v \rangle$. A path starting at a given vertex v_0, going through every other vertex exactly once, and finally returning to v_0 is called a *tour*. The length of a tour is the sum of the lengths of the edges on the path defining the tour. We are concerned with finding a tour of minimum length. A greedy way to construct such a tour is: let (P, v) represent the path so far constructed; it starts at v_0 and ends at v. Initially P is empty and $v = v_0$, if all vertices in G are on P, then include the edge $\langle v, v_0 \rangle$ and stop; otherwise include an edge $\langle v, w \rangle$ of minimum length among all edges from v to a vertex w not on P. Show that this greedy method doesn't necessarily generate a minimum-length tour.